ARL-SR-0370 ● APR 2017

US Army Research Laboratory

OS Friendly Microprocessor Architecture

by Patrick Jungwirth and Patrick La Fratta

ARL-SR-0370 ● APR 2017

US Army Research Laboratory

OS Friendly Microprocessor Architecture

by Patrick Jungwirth
Computational and Information Sciences Directorate, ARL

Patrick La Fratta
Aviation and Missile Research, Development, and Engineering Center, Redstone Arsenal, AL

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* April 2017	**2. REPORT TYPE** Special Report	**3. DATES COVERED (From - To)** September 2014–August 2016
4. TITLE AND SUBTITLE OS Friendly Microprocessor Architecture		**5a. CONTRACT NUMBER**
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Patrick Jungwirth and Patrick La Fratta		**5d. PROJECT NUMBER**
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) US Army Research Laboratory ATTN: RDRL-CIH-S Aberdeen Proving Ground, MD 21005-5067		**8. PERFORMING ORGANIZATION REPORT NUMBER** ARL-SR-0370
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		**10. SPONSOR/MONITOR'S ACRONYM(S)**
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
Patrick La Fratta is now affiliated with Micron Technology, Inc., Boise, Idaho.

14. ABSTRACT
We present an introduction to the patented Operating System Friendly Microprocessor Architecture (OSFA). The software framework to support the hardware-level security features is currently patent pending. We are interested in information technology and computer security professionals reviewing the hardware-level security features and information assurance features.
Conventional microprocessors have not tried to balance hardware performance and OS performance at the same time. The goal of the OSFA is to provide a high-performance microprocessor and OS. Computer security features are implemented in hardware. By extending Unix file permissions bits down to each cache memory bank and memory address, the OSFA provides hardware-level information assurance. OS-level access to memory is divided into access layers. For each software application, a table (white list) sets limits for all OS library function calls required by the application. Each library function call has a set of object limits. The cache bank memory pipeline architecture and permission bits provide features to balance the complexities of hardware, software, and computer security.

15. SUBJECT TERMS
microprocessor, operating system, context switch, hardware computer security, computer security, cache bank pipeline, library call permissions

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Patrick Jungwirth
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified	UU	56	**19b. TELEPHONE NUMBER (Include area code)** 410-278-6174

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

List of Tables

Preface

The paper “OS Friendly Microprocessor Architecture: Hardware Level Computer Security” was originally published in *Proceedings of SPIE: Cyber Sensing 2016, 0277-786X, V. 9826* (2016 April 19, Baltimore, MD). This report is a longer version of the published paper and it includes additional material, including 1) a bus architecture introduction, 2) Operating System Friendly Microprocessor Architecture (OSFA) Version 2 pipeline state cache bank, 3) debugging traps, and 4) architecture features for a hypervisor.

Acknowledgment

The author wishes to thank The US Army Aviation and Missile Research, Development, and Engineering Center and the US Army Research Laboratory for the opportunity to develop and improve the OS Friendly Microprocessor Architecture.

1. Introduction

The Operating System (OS) Friendly Microprocessor Architecture's (OSFA's) goals are to provide a high-performance microprocessor and reduce the code complexity of an operating system. We have developed a computer architecture that reduces the high cost of a context switch and provides hardware-based computer security. A context switch can be as fast as 1 central processing unit (CPU) cycle.

Figure 1 introduces the OSFA.[1–2] The processor memory and bus architecture is an extended Harvard architecture. The OSFA[1] uses pipeline memory controllers to rapidly background switch cache memory pages. The pipeline memory architecture supports hardware-based OS context switches. Context switches for lightweight threads can be as fast as 1 CPU cycle.

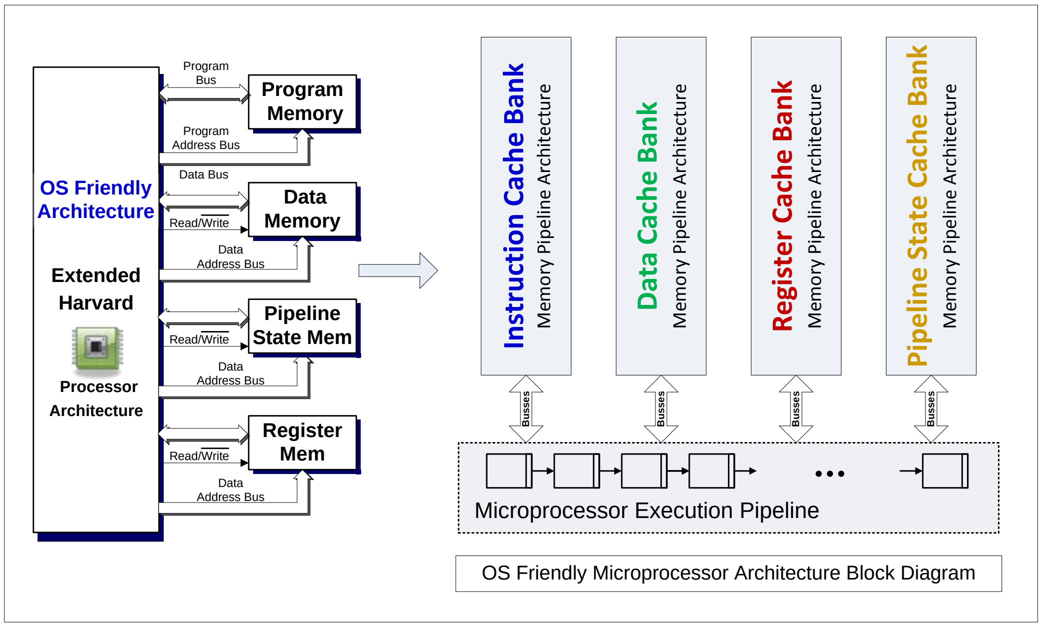


Fig. 1 OS friendly microprocessor architecture

OS information assurance is implemented in hardware. By extending the traditional Unix file permissions bits down to each memory cell, each cache line, and each cache memory bank, the OSFA processor provides hardware-based computer security.

1.1 OS Friendly Microprocessor Architecture Permission Bits

A unique feature of the OSFA is the permission bit `Index_Register_I/O` (`IRegIO`). `IRegIO` allows the OS to provide an index register pointing to an input/output (I/O) port or I/O memory address. The `IRegIO` bit "locks out" the memory address pointer (index register) from being read, written to, or modified. The running process is prevented from accessing the contents of the register; however, the process can use the index register (pointer) to read/write to I/O (registers, ports, or addresses).

The hardware permission bits can be set to allow real-time software debugging. Program debugging can use the **R** W M permission bits (Read = allowed, Write = not allowed, and Modify = not allowed) to trap all writes made to a memory address or register. This allows for hardware level debugging with zero performance overhead at the software level until a write occurs.

Library function protection is provided by extending the principal of least privilege to library function calls. For each software application, a table sets limits (white list) for all OS function calls required by the application. The library function call table sets limits for typical load, moderate load, and maximum load. Exceeding the limits for typical load, moderate load, and/or maximum load can be set to generate an exception or require higher than user level privileges.

Sections 2 through 5 cover the OSFA. Section 6 covers computer security, information assurance, and permission bits.

1.2 Bus Architectures

There are 2 commonly used microprocessor bus architectures. The von Neumann architecture consists of a unified instruction (program) and data memory. The combined memory contains both data and instructions. Newer microprocessors incorporate a no-execute bit in cache memory tables to prevent data from being executed. A Harvard bus architecture has separate instruction (program) memory and data memory. A modified Harvard architecture has internal separate caches for instructions (program) and data with a combined (unified) external memory. Figure 2 compares von Neumann and Harvard bus architectures. Note, the Harvard architecture allows for parallel memory operations over the 2 busses and memories.

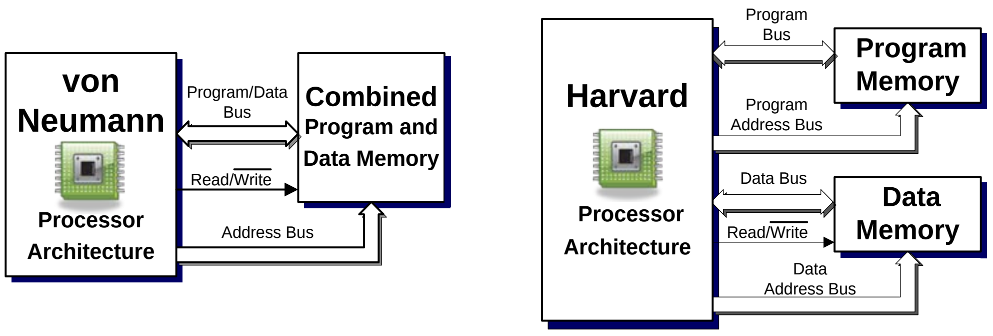


Fig. 2 von Neumann and Harvard bus architectures

Figure 3 compares computer memory types and approximate memory sizes for 2013. Register memory is the fastest memory inside a computer. Register memory typically is small. Level 1 (L1) and level 2 (L2) memory caching are contained on-chip inside the microprocessor. Level 3 (L3) memory caching can be on or off chip. Main memory is present on the main or system computer board. Hard drives and tape backups represent mass storage memory. The memory types from register to mass storage span a range of approximately 10^{11} or more.

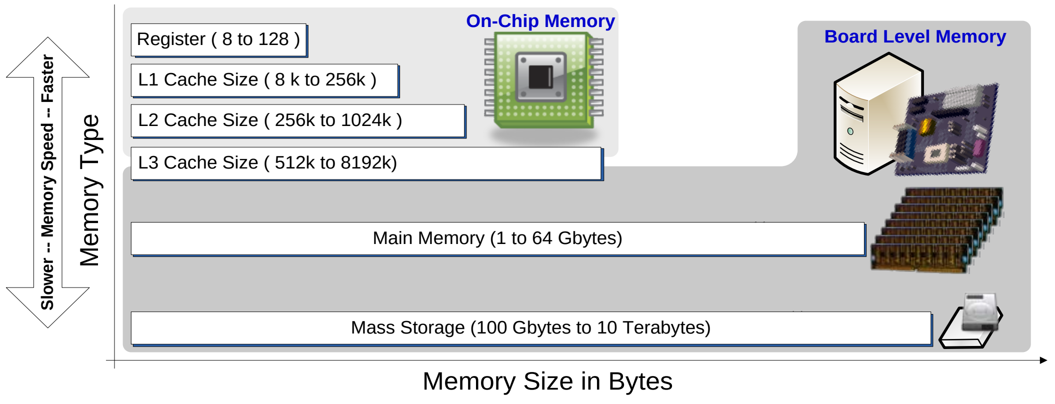

On-Chip Memory
Board Level Memory
Register (8 to 128)
L1 Cache Size (8 k to 256k)
L2 Cache Size (256k to 1024k)
L3 Cache Size (512k to 8192k)
Main Memory (1 to 64 Gbytes)
Mass Storage (100 Gbytes to 10 Terabytes)
Slower -- Memory Speed -- Faster
Memory Type
Memory Size in Bytes

Fig. 3 Computer memory types and sizes

2. In-Band Signaling, the Open Front Door

In-band signaling is an open front door. There is no user authentication for control information. A black hat or prankster only needs the tools to provide the in-band control signals to the network system.

Telephone in-band signaling combines voice (data) and control information on a telephone line. The papers by Weaver and Newell[3] and Breen and Dahlbom[4] provided the technical details for controlling the telephone network. In-band signaling provides the open front door to send control information over the phone line. Back in the 1970s, before the telephone companies switched to out-of-band signaling, a blue box generated the control tones (codes) to control the telephone network. A “blue box”[5] built by Steve Wozniak is on display at the Computer History Museum.[6] The average electronics hobbyist could easily build a blue box. Blue box phone calls were free. It did not take long for “free” blue box phone calls to become illegal.

The classic buffer overflow error, unfortunately all too common in modern programming, presents an opportunity for a black hat to place control information inside and gain control of a computer. The control information could be a line of code to jump to a computer virus or other malware application.

Caller ID does not have any authentication. A prank caller can easily spoof Caller ID. Caller ID uses a 1200 Hz frequency shift keying, Bell 202 modem[7] to send caller ID information. An “orange box” generates the spoofed Caller ID string[8] for the telephone network. In-band signaling is an open front door for controlling, spoofing, and/or hacking a system.

The OSFA’s information assurance goal is to completely separate control and data at the hardware level. The objective is to raise the difficulty level to hack a computer system. Keep in mind that claiming a system is unhackable is like creating an unsinkable ship. Current computer security best practices are based on a risk analysis and cost/benefit analysis.

3. OS Friendly Microprocessor Architecture

This section describes the OSFA’s cache bank architecture. Section 5 covers the cache bank and memory cell hardware permission bits.

The OSFA uses an extended Harvard architecture as illustrated in Fig. 4. In a Harvard architecture (see Fig. 2), there are separate busses and memories for instructions (programs) and data. The OSFA uses 4 separate busses and memories for high-speed context switching and hardware-level information assurance. A modified extended Harvard architecture has a unified external memory with separate internal caches. A context switch only requires cache banks to be connected and disconnected to the execution pipeline. Cache bank contents are background copied to and from L1 caching while the execution pipeline is running another process or thread.

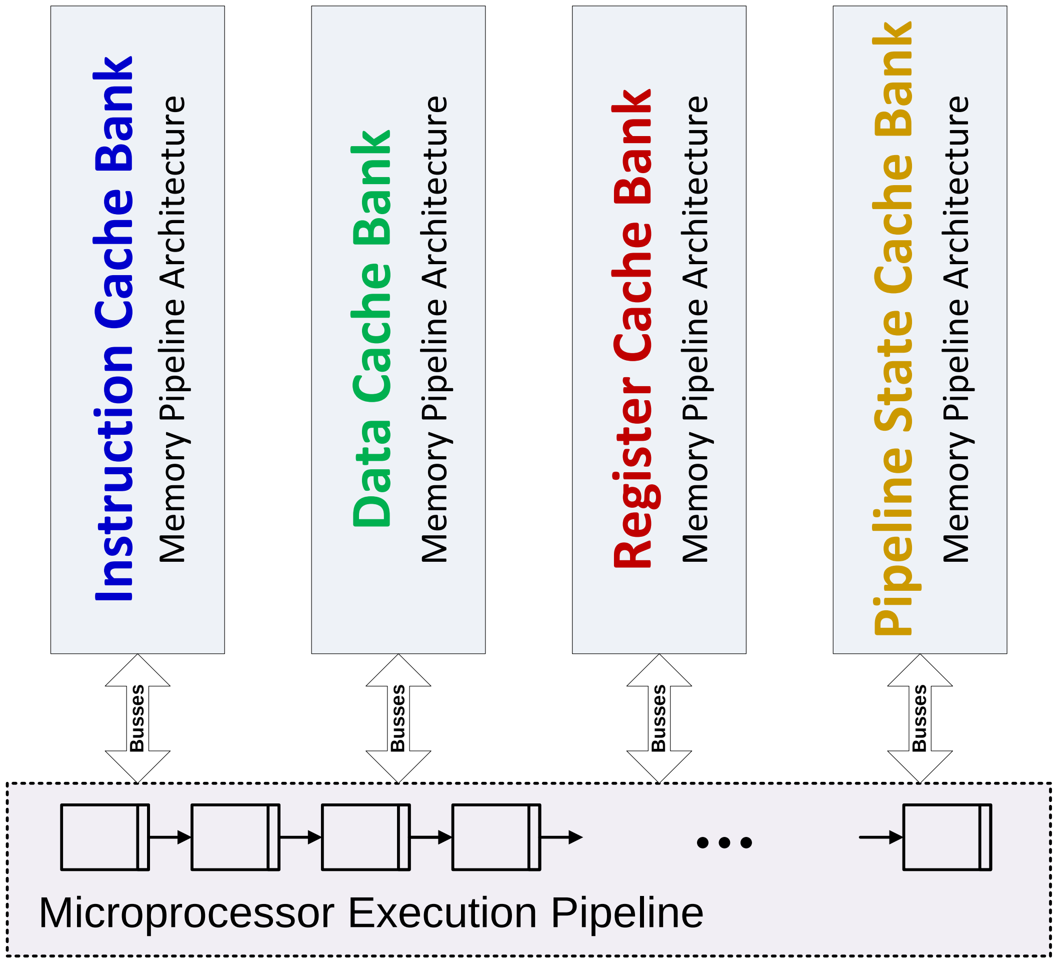
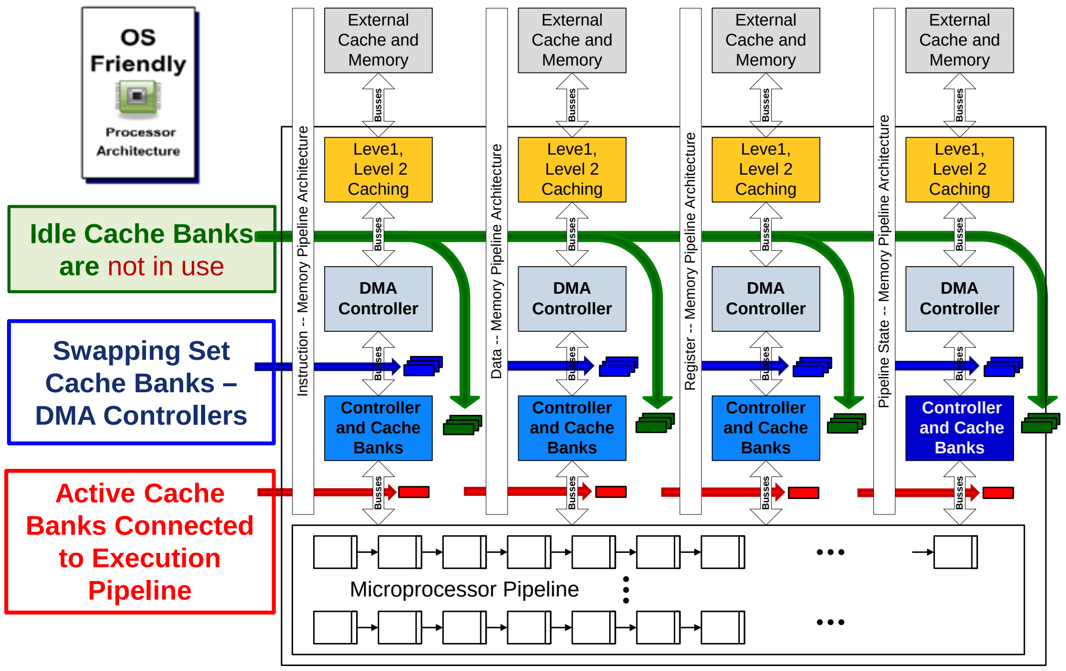

Instruction Cache Bank
Memory Pipeline Architecture
Data Cache Bank
Memory Pipeline Architecture
Register Cache Bank
Memory Pipeline Architecture
Pipeline State Cache Bank
Memory Pipeline Architecture
Busses
Busses
Busses
Busses
Microprocessor Execution Pipeline

Fig. 4 OS friendly microprocessor architecture

3.1 DMA/Cache Bank Controller Architecture

The OSFA in Fig. 5 consists of 4 DMA/cache controller banks (Instruction, Data, Register, and Pipeline State), connected to a microprocessor execution pipeline. The OSFA is a set of memory blocks (stages) in a pipeline configuration. The DMA/cache controller banks (instruction, data, register, and pipeline state) connect to internal level 1/level 2, and such, caching through busses. Internal caches connect to external caches and external memories. The OSFA can also use a unified external memory architecture similar to a modified Harvard architecture (internal separate caches for instructions and data, and a unified external memory).



OS Friendly Processor Architecture
Idle Cache Banks are not in use
Swapping Set Cache Banks – DMA Controllers
Active Cache Banks Connected to Execution Pipeline
Instruction -- Memory Pipeline Architecture
Data -- Memory Pipeline Architecture
Register -- Memory Pipeline Architecture
Pipeline State -- Memory Pipeline Architecture
External Cache and Memory
Leve1, Level 2 Caching
DMA Controller
Controller and Cache Banks
Busses
Microprocessor Pipeline

Fig. 5 OS friendly DMA controller and cache bank controller pipeline architecture

The instruction, data, and register cache bank controllers are configured to only write one block at a time from/to the processor pipeline. There is a tradeoff between cache bank size and writing data in parallel. The DMA/cache bank controllers use a parallel bus to copy to (L1 and L2 caches, internal/external L3 caches, and main memory) memory. The pipeline caching structure also allows the execution pipeline to run at full speed while hardware controllers provide background cache to memory (L1 and L2 caches, internal/external L3 caches, and main memory) copy operations in parallel.

In version 1 of the OSFA,[1] the pipeline state controller and cache bank is fully parallel. For instruction, data, and register DMA/cache controller banks, cache memory size is more important than a fully parallel memory copy. Version 2 of the OSFA [2] merges the pipeline state cache banks with the execution pipeline. A parallel memory copy is not required in Version 2 [2] since the cache banks are already stored in the execution pipeline stages.

3.2 Context Switch

A typical process is allowed to run for milliseconds before context switching to the next process. As long as the instruction, data, register, and pipeline state DMA controller/cache memory banks in Fig. 5 can complete background copy operations on the order of milliseconds, the processor does not “see” any of the background operations. Since instruction, data, register, and pipeline state memory for L1, L2, L3 caching, and external main memory can now run at a lower clock frequency, significant power savings results without decreasing processor performance.

Figure 6 illustrates the OSFA Version 1[1] write (copy) and read (load) cache bank operations for processes *n* – 1, *n*, and *n* + 1. OSFA Version 2[2] removes the parallel memory copy (see Section 4.3.2). We use sequential numbers to simplify Fig. 6. In an actual system, the process identification (PID) numbers would be arbitrary. The load (read) and write (copy) cycles for each cache bank (instruction, data, register, and pipeline state) are shown. This is a worst-case example showing cache writes and loads for each context switch. Data locality would limit the number of writes and loads resulting in more time to copy memory, allowing for more power savings. Fully parallel memory copy operations, for the pipeline state cache bank, are shown (OSFA Version 1). The "Bank(*m*)" notation refers to cache bank number (*m*) or a set of bank numbers (*m*)'s. The instruction, data, register, and pipeline state cache controller banks consist of cache banks in 1) **act**ive use by the execution pipeline: instruction.**act**, data.**act**, register.**act**, and pipeline_state.**act**; 2) **swap**ping set cache banks (instruction.**swp**, data.**swp**, register.**swp**, pipeline_state.**swp**) in use by instruction, data, register, and pipeline state DMA controllers as illustrated in Figs. 5 and 6; and 3) **ina**ctive cache banks: instruction.**ina**, data.**ina** register.**ina**, and pipeline_state.**ina** not in use by execution pipeline and not in use by DMA controllers in Fig. 5.

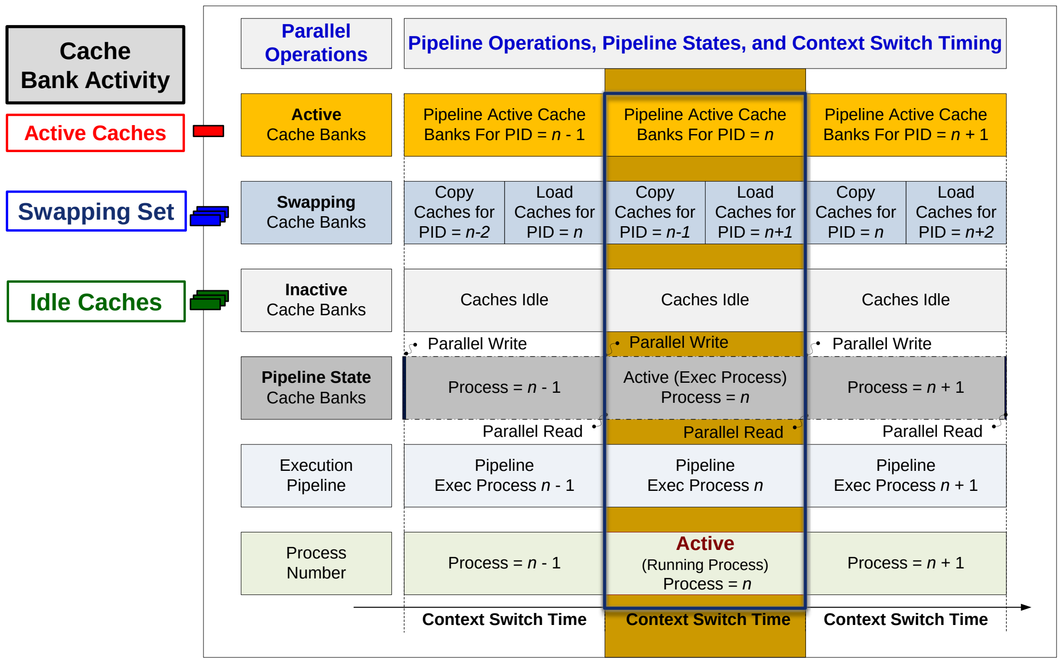


Fig. 6 OS friendly microprocessor architecture context switch timing diagram

At process *n*'s start, the active pipeline state cache bank (pipeline_state.**act**) is copied in parallel (OSFA Version 1) into the execution pipeline latches. At the end of context for process *n*, the pipeline state latches (OSFA Version 1) are copied in parallel to the active pipeline state cache bank (pipeline_state.**act)**. During context time for process *n*, the inactive cache banks instruction.**ina**, data.**ina**, register.**ina**, and pipeline_state.**ina** are idle. For process *n* – 1, the swapping set cache banks instruction.**swp**, data.**swp**, register.**swp**, and pipeline_state.**swp** are copied to L1 level caching as shown in Figs. 5 and 6. The swapping set cache banks currently in L1 cache memory, instruction, data, register, and pipeline_state, for process *n* + 1 are loaded into cache banks instruction.swp(*n* + 1), data.swp(*n* + 1), register.swp(*n* + 1), and pipeline_state.swp(*n* + 1), to prepare to execute process *n* + 1 during the next context time.

At end of context for process *n*, the active process *n* cache banks are set to swapping set cache banks: instruction.swp(*n*) = instruction.act, data.swp(*n*) = data.act, register.swp(*n*) = register.act, and pipeline_state.swp(*n*) = pipeline_state.act. After context switching from process *n* to process *n* + 1, the swapping set cache banks for process *n* + 1 are set to active: instruction.act = instruction.swp(*n* + 1), data.act = data.swp(*n* + 1), register.act = register.swp(*n* + 1), pipeline_state.act = pipeline_state.swp(*n* + 1). The cache banks instruction.act, data.act, register.act, and pipeline_state.act and now in use by execution pipeline. Figures 5 and 6 illustrate how the instruction, data, register, and pipeline_state DMA controllers run in parallel with the execution pipeline.

3.3 Cache Bank Architecture

The instruction, data, and register cache bank controllers and cache banks only need to write one word (*n* bits) at a time. Conventional microprocessors have a small number of registers: on the order of 16–128. The OSFA envisions a much larger number of registers. We envision instruction and data cache banks on the order of 128,000 or larger and register cache banks on the order of 1000 or larger. The pipeline state cache bank is on the order of 128–1000. Figure 3 compares the sizes of memories and caches for conventional architectures.

3.3.1 OS Friendly Microprocessor Architecture Version 1 Cache Bank

For OSFA Version 1, the pipeline state cache controller and cache banks need to be able to read or write to all of the pipeline stage latches in parallel. Figure 6 illustrates, the parallel load (read) and write operations for the pipeline state cache controller and cache banks. OSFA Version 2, in Section 4.3.2, removes the parallel read/write required for Version 1.

Figure 7 shows the cache controller and cache bank architecture for the instruction, data, and register banks for OSFA Versions 1 and 2. The bank selection controller provides arbitration to prevent the DMA controller and microprocessor execution pipeline from accessing the same cache bank at the same time. This separation allows the DMA to transfer cache memory pages to L1 caching in the background while the microprocessor pipeline is executing instructions. The bank address controller sets the cache bank memory addresses for the swapping set cache banks (instruction.swp, data.swp, and register.swp) and the active cache banks (instruction.act, data.act, register.act and pipeline_state.act). The read/write controllers set the data direction for the swapping set cache banks and the active cache banks.

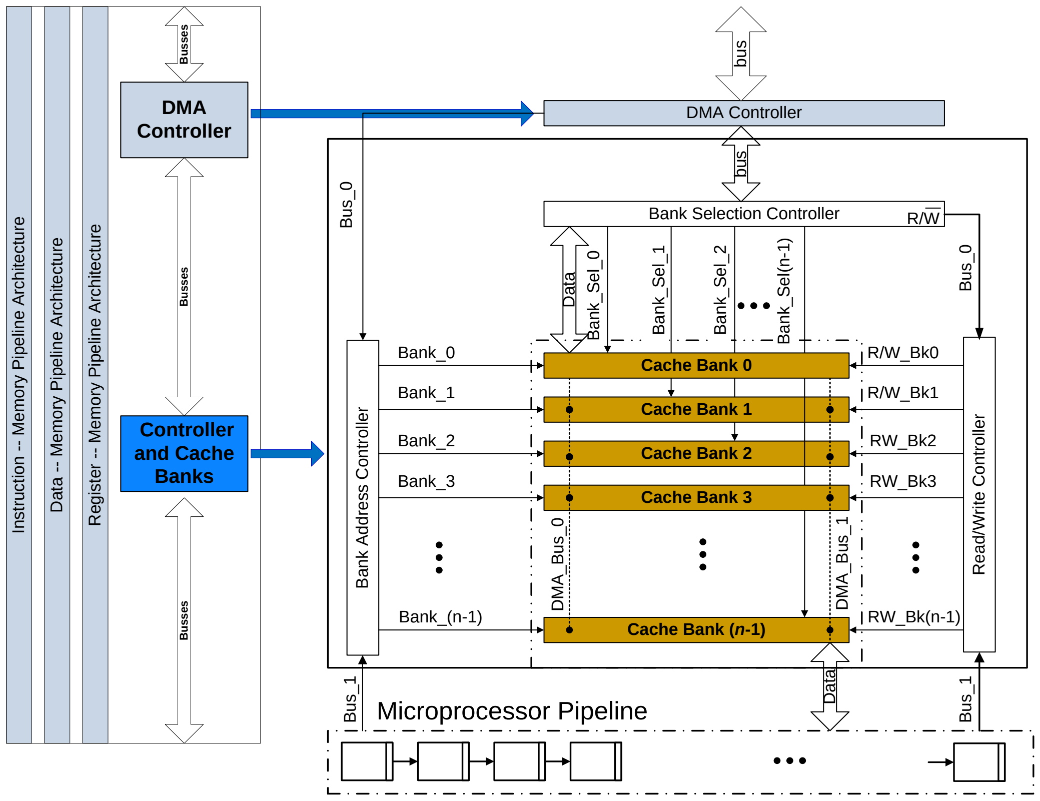
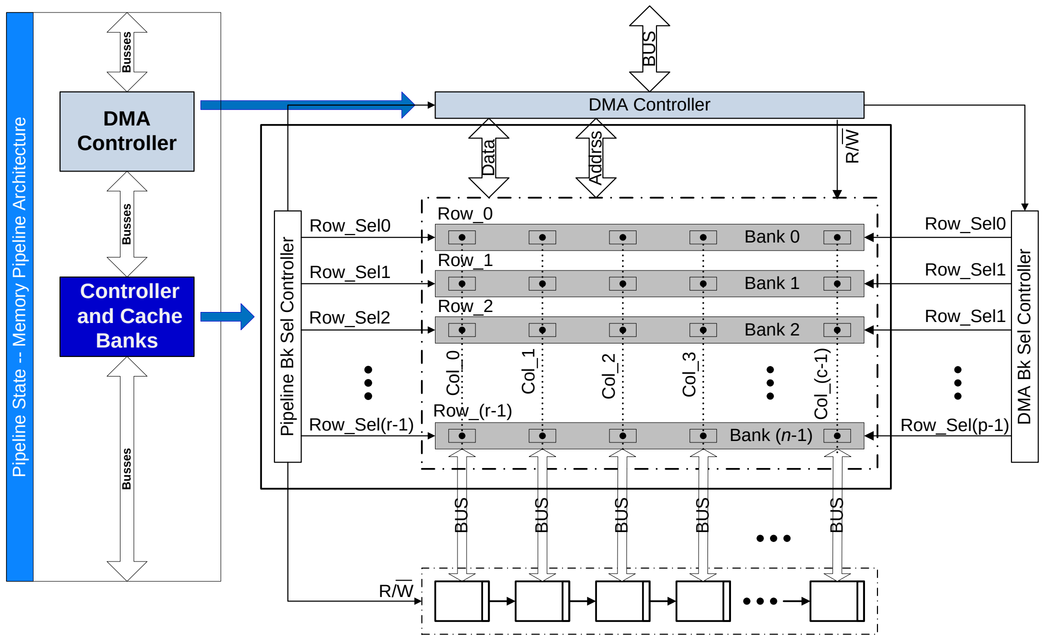


Fig. 7 Data, instruction, and register cache controller banks

Figure 8 shows the pipeline state cache controller and cache banks for OSFA Version 1. The pipeline state cache bank controller and DMA cache bank controller provide arbitration preventing the DMA controller and pipeline state (pipeline stage latches) from using the same cache bank at the same time. This separation allows the DMA to transfer a pipeline state cache memory bank to L1 caching in the background while the microprocessor pipeline is executing instructions. At the start of a context, as shown in Fig. 6, the active pipeline state cache bank (pipeline_state.**act**) is copied into the pipeline state (pipeline stage latches) in parallel in a single CPU clock cycle. At the end of a context, the pipeline state is copied in parallel in a single CPU clock cycle to the active pipeline state cache bank (pipeline_state.**act**).



Note: **pipeline_state.act ≠ pipeline_state.swp** banks; bank arbitration prevents DMA and execution pipeline from writing to or reading from the same cache bank at the same time.

Fig. 8 Pipeline state parallel cache controller banks

3.3.2 OS Friendly Microprocessor Architecture Version 2 Pipeline State Cache Bank

For the OSFA Version 2, the 8 memory latches are included in each pipeline stage as shown in Fig. 9. For example, for process *n*, `Latch4` is currently in use. To switch to process *n* + 1, `Latch4` is disconnected from the pipeline stage, and another latch, for example `Latch2`, is connected. The latches used by process *n* may now be background copied to L1 cache sequentially from stage 0 through stage (*m* – 1) (all of the pipeline latches) during context *n* + 1. The Version 2 pipeline state DMA/cache controller pipeline offers the same processor performance as Version 1 while requiring less power.

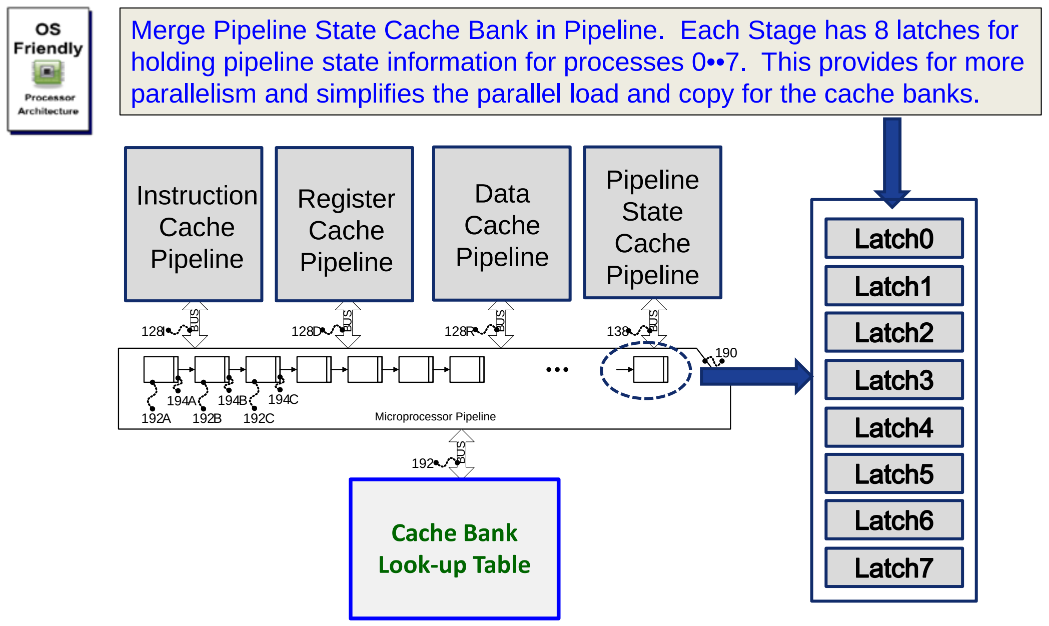

OS Friendly
Processor Architecture
Merge Pipeline State Cache Bank in Pipeline. Each Stage has 8 latches for holding pipeline state information for processes 0••7. This provides for more parallelism and simplifies the parallel load and copy for the cache banks.
Instruction Cache Pipeline
Register Cache Pipeline
Data Cache Pipeline
Pipeline State Cache Pipeline
BUS
128I
128D
128R
138
190
194A
194B
194C
192A
192B
192C
Microprocessor Pipeline
192
Cache Bank Look-up Table
Latch0
Latch1
Latch2
Latch3
Latch4
Latch5
Latch6
Latch7

Fig. 9 **OS friendly microprocessor architecture version 2 pipeline state cache banks**

3.4 OS Friendly Microprocessor Architecture Performance Modeling

The OSFA[1] offers new opportunities for increased performance and decreased power consumption by providing hardware features to reduce the OSs cost for managing resources. Sections 4.4.1–4.4.3 develop a first-order approximation of the potential improvements in OS Friendly Microprocessor Architecture's context switch performance. Conventional microprocessor performance models are based on the research from Vangal et. al.[9] and Mudge.[10]

3.4.1 Conventional and OS Friendly Microprocessor Architecture Context Switch Modeling

Sections 4.4.2 and 4.4.3 estimate the context switch time required for a conventional architecture and the new OSFA. The OSFA significantly improves the context switch time and uses less power. The high-level representations for conventional and improved OSFA architectures are shown in Figs. 10 and 11. These figures assume the following architectural characteristics. First, the internal designs of the execution pipelines in the 2 architectures are the same. The model for the OSFA execution pipeline in Fig. 11 uses the same execution pipeline as the conventional architecture in Fig. 10. The labels inside the pipeline stages (labeled "PS") refer to the stages to which the following sections reference (EX: Execution Stage, MEM: Memory Access Stage, and WB: Writeback Stage). Next, it is assumed that the pipelines in both architectures, OSFA and conventional, operate at a fixed voltage V_{P_H} with clock period t_S. The Register File Set (RFS), active register cache bank **(register.act)**, and the active pipeline state cache bank (pipeline_state.**act**) normally operate at voltage V_{R_H} with clock period t_S. For power improvements, the OSFA can dynamically scale down both the voltages and clock rates of the inactive and swapping cache controllers and cache banks. The voltage of inactive and swapping cache controllers and cache banks can be reduced to some value V_L, while the clock frequency (clock period) of these components can be reduced (clock period increased) to some value, clock frequency f_L, or clock period t_L.

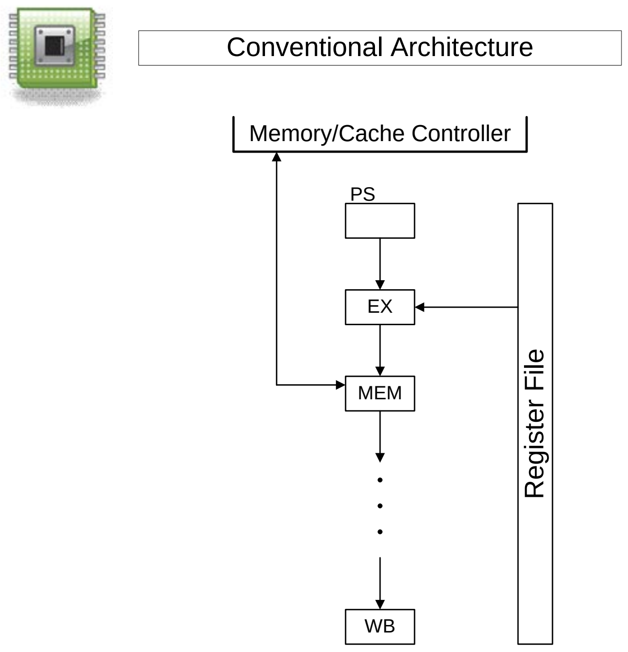


Fig. 10 Conventional processor architecture model

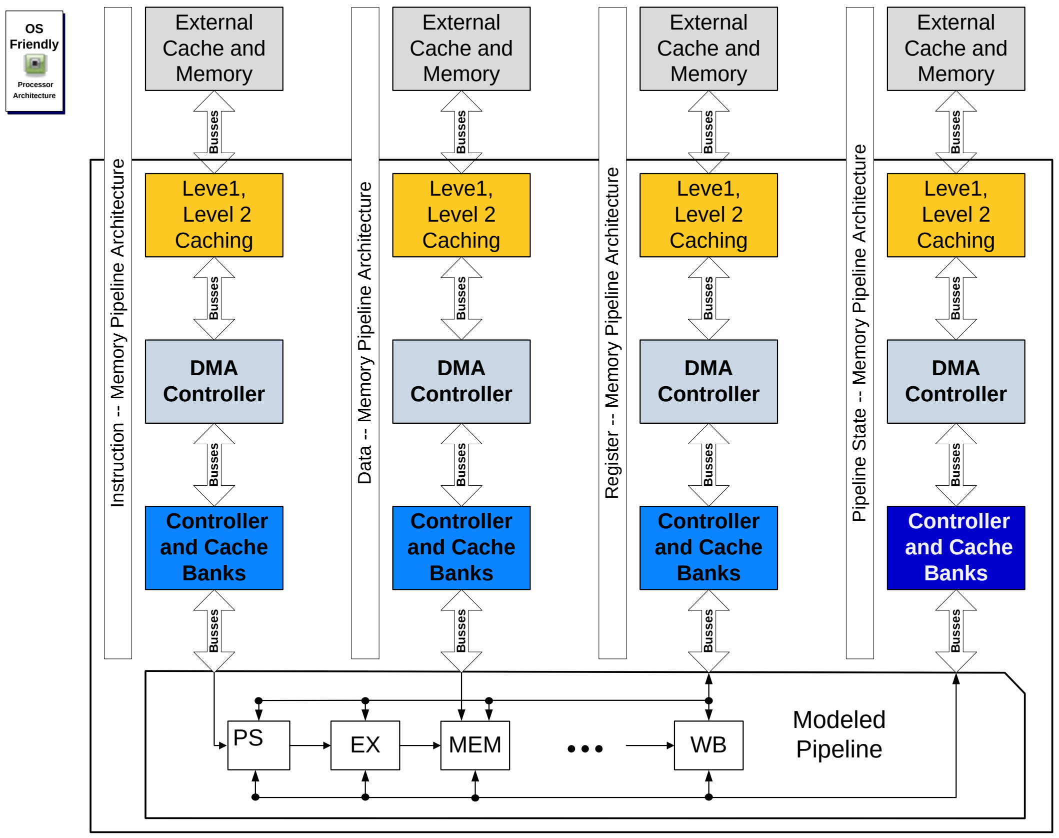

OS Friendly
Processor Architecture
Instruction -- Memory Pipeline Architecture
Data -- Memory Pipeline Architecture
Register -- Memory Pipeline Architecture
Pipeline State -- Memory Pipeline Architecture
External Cache and Memory
Busses
Leve1, Level 2 Caching
DMA Controller
Controller and Cache Banks
PS
EX
MEM
•••
WB
Modeled Pipeline

Fig. 11 OS friendly microprocessor architecture model

3.4.2 Conventional Architecture Context Switch Modeling

This section presents the steps taken by conventional processor architecture in Fig. 10 to perform a context switch. Each step requires a certain period of time, which is determined by the amount of work required by the step, the clock rate of the components involved, and the parallelism exploited by these components. All components of the conventional architecture operate with the short clock period t_S. The steps involved in a context switch for the conventional processor are shown in Table 1.

Table 1 Conventional architecture context switch steps

Step	Description
Step 1	Flush the pipeline state out to the register file.
Step 2	Write out each register value to memory.
Step 3	Bring the OS register state back into the register file.
Step 4	Refill the pipeline with the OSs pipeline state.
Step 5	Execute the standard OS operations.
Step 6	Flush the OS pipeline state to the register file.
Step 7	Write out each register value to memory.
Step 8	Bring the register state of another process, *p*, back into the register file.
Step 9	Refill the pipeline with *p*'s state.

Assuming the conventional pipeline in Fig. 10 has *s* stages, step 1 will require *s* clock ticks, and hence $s \cdot t_S$ time. Step 2, writing each register file out to memory, requires reading each register value into the EX stage, moving it into the MEM stage, and then flushing it out to memory. There are 3 clock ticks for each register value, but since the operations can be performed in a pipelined fashion, we approximate this as $r \cdot t_S$ time total for all *r* registers. Step 3 requires filling up the pipeline to retrieve register values from memory, requiring *s* ticks, then writing each value back to the register file in the writeback stage for a total of $(s + r) \cdot t_S$ time. Step 4 is filling the pipeline back up with values from the register file, but this can be pipelined with the register file refill and hence is already accounted for. Step 5 takes some unknown amount of time, t_{OS_NORMAL}, that is dependent on the OS design. Steps 6 and 7 are similar to steps 1 and 2, which again require $s \cdot t_S$ time and $r \cdot t_S$ time, respectively. Step 8 is like step 3, which requires $(s + r) \cdot t_S$ time, and step 9 is like step 4, which is accounted for in this time. Hence, an expression that approximates this entire process is given by Eq. 1 and simplified in Eq. 2.

$$t_{CS_{CONV}} = st_S + rt_S + (s+r)t_S + t_{OS_{NORMAL}} + st_S + rt_S + (s+r)t_S. \qquad (1)$$

$$t_{CS_{CONV}} = 4t_S(r+s) + t_{OS_{NORMAL}} \quad \text{(Conventional Architecture's Context Switch Time)} \quad (2)$$

3.4.3 OS Friendly Microprocessor Architecture Context Switch Modeling (Version 1)

Figure 6 presents a worst-case timing diagram for the OSFA Version 1 assuming swapping set cache banks (**instruction.swp**, **data.swp**, **register.swp**, and **pipeline_state.swp)** must be loaded and written for every context switch. Data locality will significantly reduce the number of cache bank memory copy operations. The model for OSFA's execution pipeline in Fig. 11 is same as the conventional architecture described in Section 4.4.2 and Fig. 10. A more optimized pipeline would provide higher performance.

The OSFA pipeline model also operates with clock period t_S. In the determination of the clock frequency of the OSFA's other components, the cache banks are divided into 3 sets: **act**ive, **ina**ctive, and **sw**a**p**ping set. One of the register cache banks, register.**act,** is active and one of the pipeline state caches, pipeline_state.**act**, is active. These active cache banks are those that are in use by the OSFA pipeline in Fig. 4 and the modeled pipeline in Fig. 11. There is then a set of the other cache banks, **instruction.bank(*m*)**'s, **data.bank(*m*)**'s, **register.bank(*m*)**'s and **pipeline_state.bank(*m*)**'s, that are either flushing state out to the DMA controllers (instruction DMA, data DMA, register DMA, pipeline state DMA) or bring state back from the DMA controllers (instruction DMA, data DMA, register DMA, pipeline state DMA). These sets are designated as the swapping sets where

> **instruction.swp =** set of **instruction.bank(*m*)**'s cache memory banks,
>
> **data.swp =** set of **data.bank(*m*)**'s cache memory banks,
>
> **register.swp =** set of **register.bank(*m*)**'s cache memory banks, and
>
> **pipeline_state.swp =** set of **pipeline_state.bank(*m*)**'s cache memory banks.

The cache banks not in use by the execution pipeline or DMA controllers are **ina**ctive or idle.

The active components instruction.act, data.act, register.act and pipeline_state.act operate with clock period t_S, the swapping components instruction.swp, data.swp, register.swp and pipeline_state.swp operate with the longer clock period t_L , and the inactive components instruction.ina, data.ina, register.ina and pipeline_state.ina are idle (for static memory, clock frequency could be set to 0 Hz).

The modeled OSFA Version 1 in Fig. 11 performs the following steps in Table 2 during a context switch. The key feature of the OSFA is that parallelism takes place at various levels to reduce execution time. In step 1, all pipeline stages flush state to the active pipeline state cache simultaneously (see Figs. 5, 6, and 11), and hence this requires only one tick at the high clock rate for a time of t_S.

Table 2 OS friendly microprocessor architecture version 1 context switch steps

Step	Description
Step 1	Flush the pipeline state (pipeline stage latches) out to the active pipeline state cache.
Step 2	Switch the active cache banks to the OS state
Step 3	If necessary (if free slots in the pipeline_state cache bank and register cache bank are needed), flush the contents of the previous process' state cache banks for the previous process ID (PID) as described in Figure 6 .
Step 4	Bring the OSs pipeline state back into the pipeline from the pipeline state cache.
Step 5	Execute the standard OS operations.
Step 6	Flush the pipeline state out to the active pipeline state cache **pipeline_stage.act**.
Step 7	If necessary, fetch the state of the next process for execution from memory into the next process' cache banks.
Step 8	Switch the active cache banks to the caches containing new (next) process (for example, next PID): **pipeline_state.act = pipeline_state(next PID), register.act = register(next PID)**, **instruction.act = instruction(next PID)**, and **data.act** = **data(next PID)**.
Step 9	Parallel copy the contents of the active pipeline state cache back into the pipeline stage latches. Section 4.4.3 describes the parallel copy for pipeline state cache controller and pipeline state cache banks.

Step 2 also takes a single tick to switch to the set of active cache banks for the next PID: instruction.act = instruction(next PID), register.act = register(next PID), data.act = data(next PID), and pipeline_state.act = pipeline_state (next PID).

Step 3 takes *s* ticks for the pipeline state cache and *r* ticks for the register file. However, these steps can be completed at the same time as steps 4–6, so as long as they are completed in at most the time for those steps, the pipeline will not see them. It is reasonable to assume that step 3 can be completed in less time (if, for the time being, we ignore cache misses and contention), as the pipeline state and register file are relatively small, while the OS must generally perform several system operations before switching back to a user-level process.

Step 4 is the reverse of step 1, so it requires only a single tick.

Step 5 still takes t_{OS_NORMAL} as with the conventional architecture, and step 6 takes a single tick like step 1. Step 7 is the reverse of step 3 and requires the same amount of time. Again, these steps can be performed in parallel with those of steps 4–6.

Step 8 is the same as step 2, and step 9 is the same as step 4. Each of these takes one tick. Hence, the total time for the OSFA context switch, t_{CS_OSFA}, is found in Eq. 3 and simplified in Eq. 4.

$$t_{CS\_OSFA} = t_S + t_S + t_S + t_{OS\_NORMAL} + t_S + t_S + t_S. \quad (3)$$

$$t_{CS\_OSFA} = 6t_S + t_{OS\_NORMAL}. \quad (4)$$

We will ignore the t_{OS_NORMAL} term by assuming it is the same for conventional and OSFA. The speedup offered by the OSFA for context switching is estimated to be $CT\_Speedup_{OSFA}$ in Eq. 5. For example, for a 5-stage pipeline, $s = 5$, and 32 general-purpose registers, $r = 32$, this translates to an estimated theoretical speedup of 25 found in Eq. 5 for OSFA. This is a significant order of magnitude speedup improvement for the OSFA compared with the conventional processor architecture.

$$CT\_Speedup_{OSFA} \approx \frac{4st_S+4rt_S}{6t_S} \approx \frac{2}{3}(s+r) = \frac{2}{3}(5+32) = 25 \quad \text{For OSFA Context Switch.} \quad (5)$$

In Eq. 6 for a large number of registers, $r \gg s$, and for $t_{OS\_NORMAL} \gg 6t_S$, with $t_S \lesssim \frac{1}{100\text{ MHz}}$, the speedup is order the number of registers, $\mathcal{O}(r)$.

$$CT\_Speedup_{OSFA} = \frac{4st_S+4rt_S+t_{OS\_NORMAL}}{6t_S+t_{OS\_NORMAL}} \approx \frac{4st_S+4rt_S+t_{OS\_NORMAL}}{t_{OS\_NORMAL}} \approx \frac{4r}{t_{OS\_NORMAL}} \approx \mathcal{O}(r). \quad (6)$$

3.4.4 OS Friendly Microprocessor Architecture Context Switch Modeling (Version 2)

OSFA Version 2 pipeline state cache bank in Fig. 9 has the same context switch speedup found in Eq. 5. The parallel memory copy for version 1 was replaced by a background serial memory copy as described in Section 4.3.2. The serial memory copy only requires a low-speed clock. Power requirements for the serial memory copy are less than the full parallel memory copy used in version 1 described in Section 4.4.3.

4. OS Friendly Microprocessor Architecture Hardware Computer Security

OS information assurance for "data" (instruction cache banks, data cache banks, register cache banks, pipeline state cache banks, and memory cells) is implemented in hardware. By extending the traditional Unix file permissions bits[11–13] down to each memory cell, memory cache line, and cache memory bank, the OSFA provides hardware-level information assurance. Figure 12 illustrates hardware-level information assurance hierarchy and permission bits.[1]

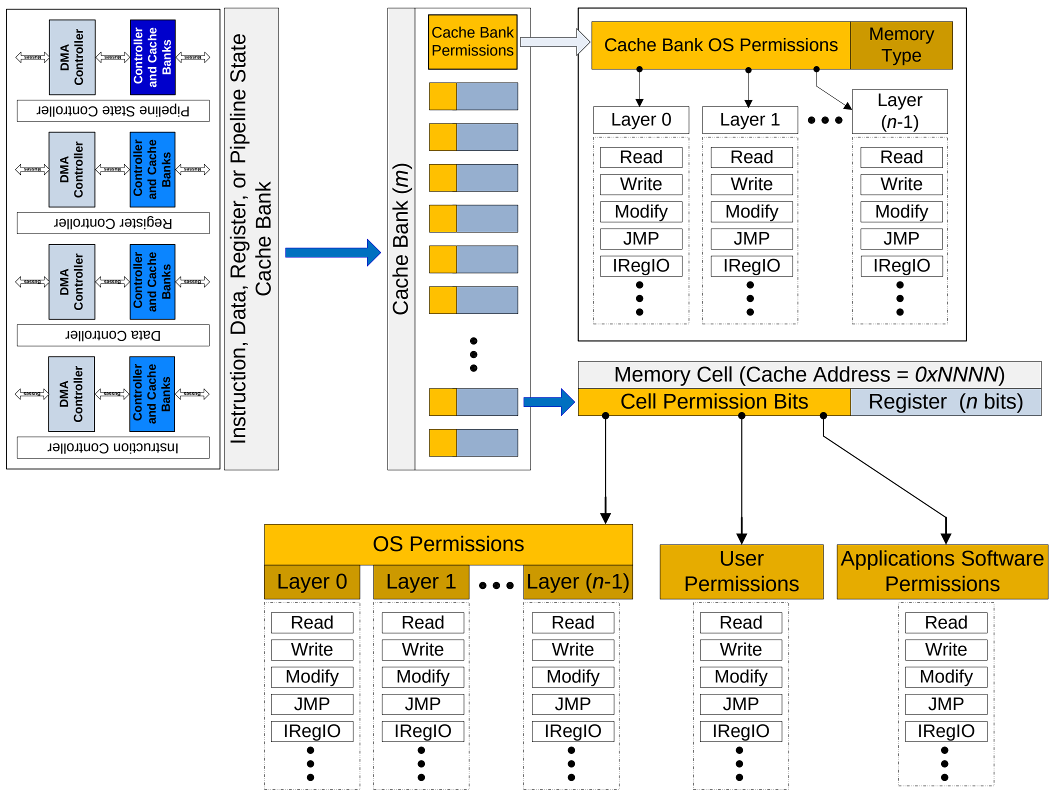


Fig. 12 Cache bank and memory cell hardware information assurance

4.1 Cache Bank and Memory Cell Permission Bits

The instruction, data, register, and pipeline state cache banks have a set of OS level permission bits. The cache bank memory type field is used to define data types (e.g., data space, stack space, heap space, integer, floating point). Only the OS has permission to access and modify cache banks' permission bits. OS level access to cache controller banks is divided into access layers (layer_0, layer_1, layer_2, etc.). Example permission bits are shown in Fig. 12. Each cache memory bank has permission bits for each memory cell. Each cache bank memory cell has permission bits for the OS layers and software (user level, and applications, etc.). The OS permission bits are further divided in OS layers (layer_0, layer_1, etc.). Additional permission bits can easily be added to Fig. 12.

4.2 Instruction Permission Bits

The OSFA also includes permission bits for additional OS level control over instructions and hardware. In Fig. 12, permission bit **JMP** provides OS level control of jump or branch on index register instructions. Permission bit `IRegIO` allows OS to provide an index register pointing to an I/O port or I/O memory address. The `IRegIO` bit locks out the index register (pointer). The running process is prevented from accessing the contents of the register/cache bank; however, the process can use the index register (pointer) to read/write to I/O (registers, ports, or addresses). Registers can be used to define data types using 2 registers for minimum value and maximum value. For example, the type `IOMemAddressType` could be defined as minimum = 0, and maximum = 15. If a register of type `IOMemAddressType` is outside the minimum/maximum range, then the processor will generate an out-of-range exception.

4.3 Library Call Permissions

The library function call table in Fig. 13 extends the principal of least privilege to the library function call level. A table listing all possible library function calls a software program may use is created. Each possible library function call is listed with typical moderate load and maximum load lower and upper limits. More limits could be used for finer grain control as in the example of the `OpenFile(•)` library function call privilege limits. If the minimum number of open files is 0, the lower limits for cases typical, moderate, and maximum is 0. If the typically user will only have 5 files open at a time, the upper limit for typical is 5. Maximum load upper limit specifies the maximum number of files that may be open at a time. Exceeding the upper limits can be set to 1) require higher than user level privileges or to 2) generate an exception. The digital signature provides authentication of the library function call table and its permission settings.

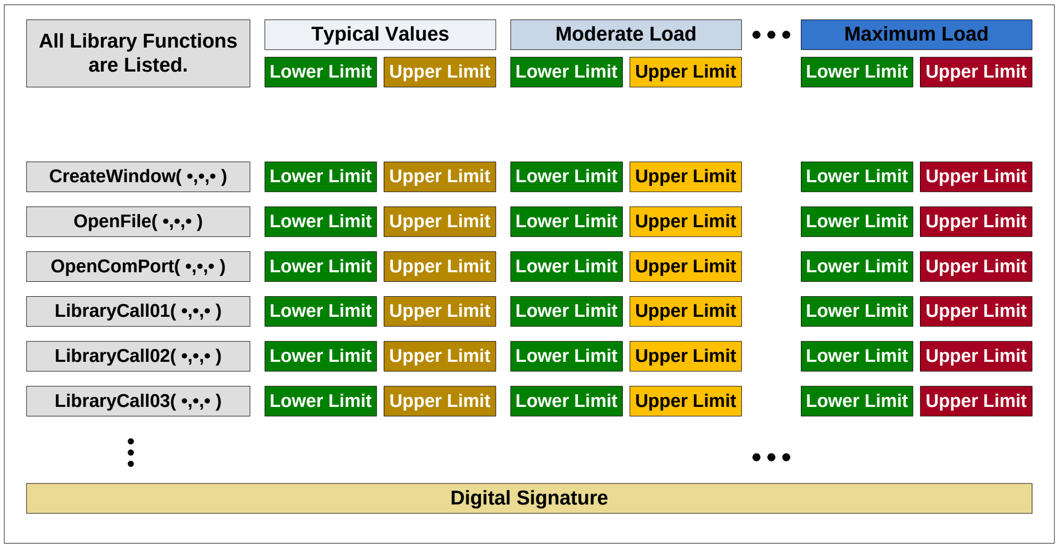


Fig. 13 Library function call table information assurance

5. OS Friendly Microprocessor Architecture Access Layers

Figure 12 introduces the cache bank permission bits for the OSFA. The instruction, data, register, and pipeline state cache banks all use the same block of permission bits. Figure 14 focuses on the cache bank permission bits. Each cache bank contains a memory type field. The memory type field can be used to define stack space, heap space, user memory, shared memory and the like. Each cache bank contains a list of permission bits for the OS rings or OS layers. One possible hierarchy for the OS Layers is found in Table 3. The secure microkernel and microkernel drivers control all permission bits and manage I/O and memory allocation.

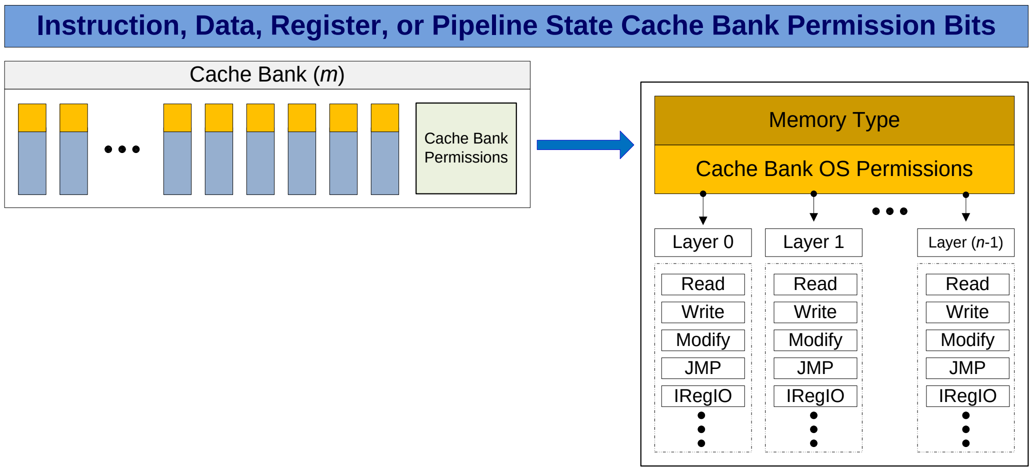

Instruction, Data, Register, or Pipeline State Cache Bank Permission Bits
Cache Bank (m)
Cache Bank Permissions
Memory Type
Cache Bank OS Permissions
Layer 0
Layer 1
Layer (n-1)
Read
Write
Modify
JMP
IRegIO

Fig. 14 OS friendly microprocessor architecture cache bank permission bits

Table 3 Example OS friendly microprocessor architecture layer hierarchy

Layer Number	Trust Level	OS Access Level
Layer_0	Complete	Secure Microkernel
Layer_1	🟩	Microkernel Drivers
Layer_2	🟨	Hypervisor
Layer_3	🟧	Thick OS
Layer_4	🟧	Dynamic Link Libraries
Layer_5	🟧	OS Drivers
Layer_6	🟫	• • •
Layer_7	Untrusted	Applications Software

As illustrated in Fig. 5, all memory, and I/O is managed as direct memory access. For example, an application writes a block of data to a hard drive. The application executes `file.create(•)`. The OS level file operation calls the secure microkernel for an I/O port. The secure microkernel passes a pointer to an I/O port. The pointer to the port address is marked R W M **`IRegIO`** (read, write, and modify are not allowed, `IRegIO` specifies pointer to I/O port). We will use the notation R W M for read, write, and modify are allowed. The OS cannot read, write, or modify the value of the pointer. The application then calls `file.write(•)` using the pointer. The OS can only use the file pointer for I/O for the `file.write(•)` OS library function call. Microkernel can manage (run) multiple OSs and programs at the same time.

5.1 Instruction, Data, Register, and Pipeline State Memory Partitions

The instruction, data, register, and pipeline state cache banks all use the same permission bits. The memory type field in Fig. 14 restricts the information that can be placed in the cache bank. For example, if the cache bank type is set to application data, the running task or thread cannot use the cache bank as stack space. For a cache bank to be used for stack space, the stack space permission bit must be set. For a push or pull stack operation to read or write to a stack cache bank, an index pointer must have stack permission bit set. Stack registers will also be marked as R W M (read, write, modify are not allowed), so the running task cannot modify the contents of the stack pointer (register).

Instructions and data have their own stack space (cache banks). For example, an array of data is placed on the stack to call an OS library function. The return address is not data and is placed on a separate stack contained in the instruction cache bank. The stack spaces are not unified. The data stack does not contain any return addresses. The instruction stack is managed by the microkernel, so the OS and application do not have any direct access to the return address pointer. The return address pointer can also be set to R W M (read, write, modify are not allowed).

The next section looks at permission bits for different OS access levels covered in Table 3.

5.2 Permission Bits: Microkernel, Thick OS, Drivers, and Applications

Figures 15–17 provide example permission bit settings. Figure 15 shows permission bit settings for the secure microkernel, the most secure layer. The microkernel has complete access. Microkernel drivers have read and write access. In Fig. 16, the full feature OS and dynamic link libraries are set to access layers 3 (OS) or 4 (DLL) through $n - 1$. OS hardware drivers are set to read and write access only. The application’s permission bits are shown in Fig. 17. The multiple levels of permission bits allow for restricting the OS from writing to applications’ areas of memory. Keep in mind that each cache bank memory can have different permission bit settings. The OS could have full control over one application's cache memory bank and have no control or visibility to a second application's cache memory bank. The large number of permission bits allows for fine grain memory access control. We could set the permission bits to allow a trusted application to run under an untrusted OS. The permission bits allow for hardware sandbox execution of unknown, untrusted code. The hardware permission bits require that we at least completely trust the secure microkernel. A formal proof of correctness or a very high assurance level microkernel is recommended.

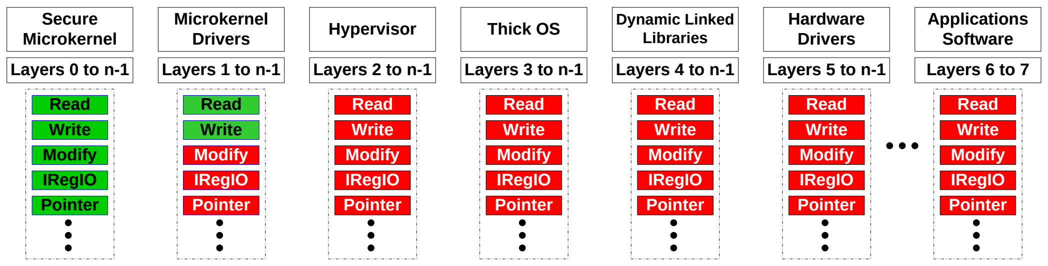


Fig. 15 Secure microkernel cache banks and permission bits

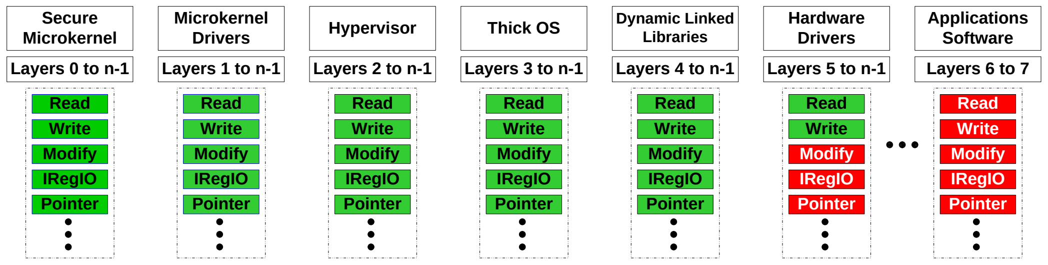


Fig. 16 Thick OS cache banks and permission bits

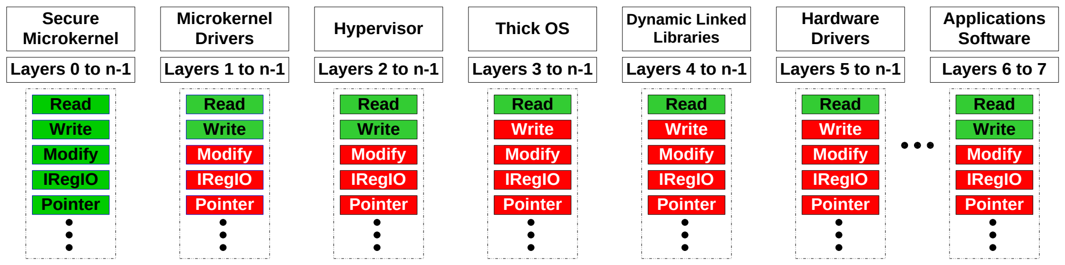


Fig. 17 Application's permission bits

5.3 I/O Implementation

The OSFA uses cache banks and direct memory access controllers for I/O. The cache bank architecture allows all I/O to have a uniform structure. In a high-level programming language, we use file operations like `file.open(•)`, `file.write(•)`, and `file.read(•)`. The file function calls provide a uniform structure abstracting the low-level details away from the programmer. By using cache banks and direct memory access for all input and output, all I/O operations at the hardware level are the same. As illustrated in Fig. 18, a USB controller interfaces to a direct memory access controller. This allows cache banks to be the basic block for all input and output operations. We could replace the USB controller with a disk drive controller or a video graphics card. The cache bank and direct memory access controller provide a universal interface between the OSFA and other devices.

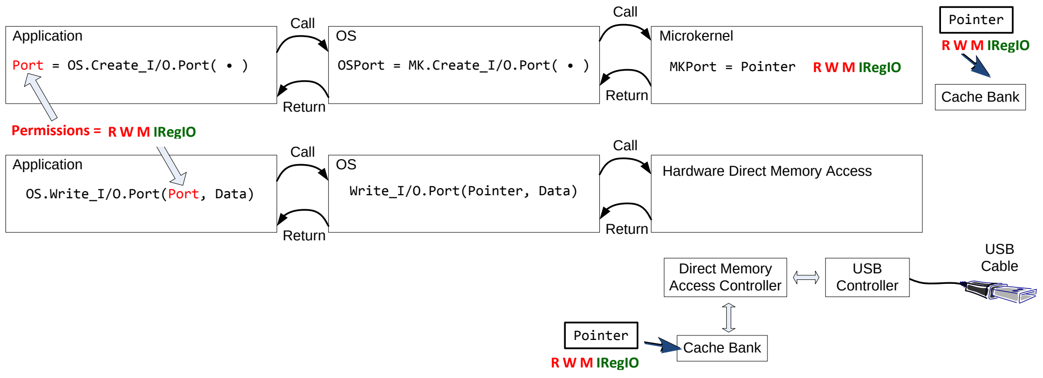

Application
Port = OS.Create_I/O.Port(•)
Call
Return
OS
OSPort = MK.Create_I/O.Port(•)
Call
Return
Microkernel
MKPort = Pointer R W M IRegIO
Pointer
R W M IRegIO
Cache Bank
Permissions = R W M IRegIO
Application
OS.Write_I/O.Port(Port, Data)
Call
Return
OS
Write_I/O.Port(Pointer, Data)
Call
Return
Hardware Direct Memory Access
Direct Memory Access Controller
USB Controller
USB Cable
Pointer
R W M IRegIO
Cache Bank

Fig. 18 OS friendly microprocessor architecture I/O example

In Fig. 18, the software applications calls the OS library function `OS.Create.I/O_Port(•)`. The OS calls the microkernel function `MK.Create.I/O_Port(•)` to create a pointer to a cache bank. The pointer has permission bits set to R W M **`IRegIO`**. The OS and applications software cannot read, write, modify, or access the contents of the index register (pointer). All the OS and application know is that the index register, for example register 10 (R10), contains a pointer to the I/O port. Figure 19 gives a real-world example for comparison. The junior engineer knows the project plan is contained in the safe; however, he does not have access to the project plan.

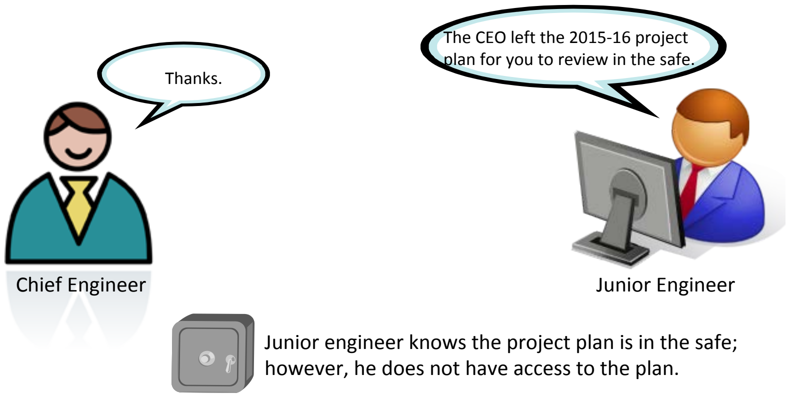

Thanks.
The CEO left the 2015-16 project plan for you to review in the safe.
Chief Engineer
Junior Engineer
Junior engineer knows the project plan is in the safe; however, he does not have access to the plan.

Fig. 19 Real-world example of OS friendly microprocessor architecture's permission architecture

The permission bits guarantee that the pointer can be trusted by the microkernel. Here trust refers to software level. Only the microkernel has access to the actual address for the pointer to the I/O Port. The software applications in Figs. 18 and 20 know the register number containing the I/O pointer; however, the software applications cannot access the contents of the register (memory address contained in the register).

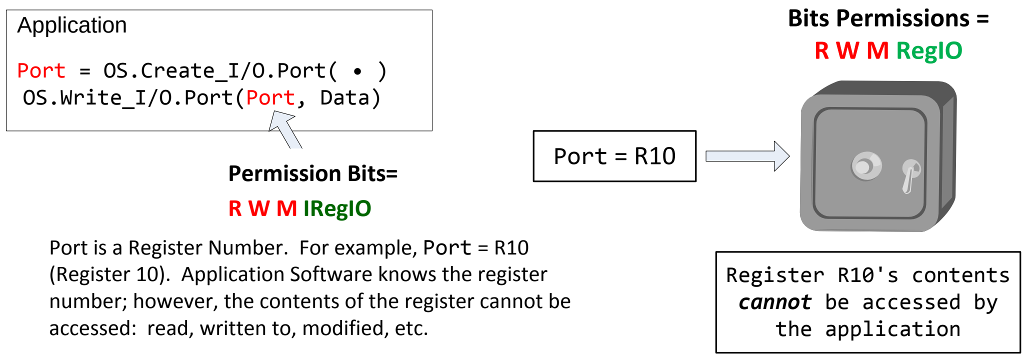

Application
Port = OS.Create_I/O.Port(•)
OS.Write_I/O.Port(Port, Data)
Permission Bits=
R W M IRegIO
Port is a Register Number. For example, Port = R10 (Register 10). Application Software knows the register number; however, the contents of the register cannot be accessed: read, written to, modified, etc.
Port = R10
Bits Permissions =
R W M RegIO
Register R10's contents cannot be accessed by the application

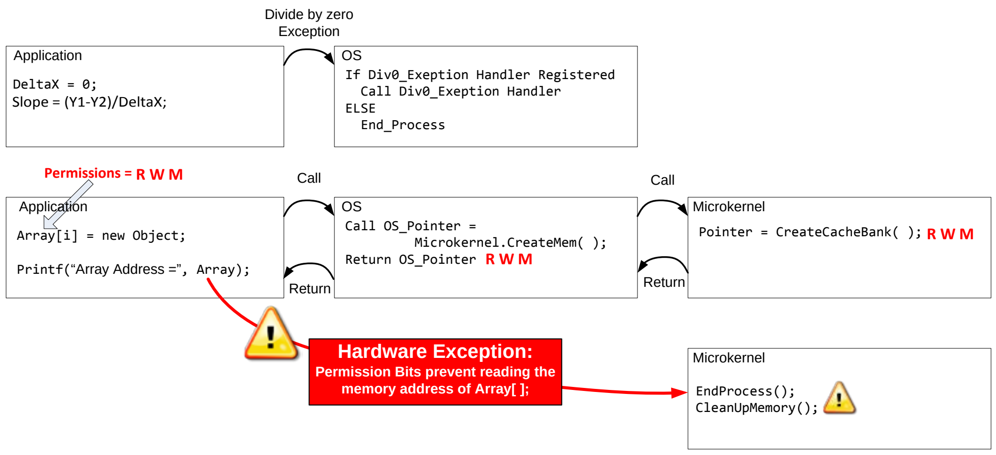
Fig. 20 For the I/O port, the application software knows the register number; however, the application cannot access the contents of the register

The application places data inside the cache bank pointed to by the pointer `Port`. The application calls `OS.Write.I/O_Port(•)`, which simply passes the pointer to the hardware DMA controller to transmit the cache bank contents. In Fig. 18, the direct memory access controller sends the contents of the cache bank to a USB controller. We have several options for the lifetime of the I/O Port pointer: 1 context time, single read/write operation, or forever for the microkernel.

In summary, the `IRegIO` permission bit restricts access to only the microkernel. The permission bits allow the microkernel to trust the pointer. An application passes the pointer to a hardware direct memory access controller to send or receive a cache bank size block of data. If a black hat had control of a process and could guess the address of a cache bank, he still would have to bypass the microkernel to set the `IRegIO` permission bit for the cache bank. The permission bits and cache bank architecture provide for efficient, high speed, low overhead I/O.

5.4 Exception Handling

Two example exceptions are illustrated in Fig. 21. A simple divide by zero error in the application software raises an exception handled by the OS. If the software application has provided an exception handler, it is called by the OS; else, the OS terminates the running application.



```
DeltaX = 0;
Slope = (Y1-Y2)/DeltaX;
```

```
If Div0_Exeption Handler Registered
  Call Div0_Exeption Handler
ELSE
  End_Process
```

```
Array[i] = new Object;

Printf("Array Address =", Array);
```

```
Call OS_Pointer =
         Microkernel.CreateMem( );
Return OS_Pointer  R W M
```

```
Pointer = CreateCacheBank( ); R W M
```

```
EndProcess();
CleanUpMemory();
```

Fig. 21 Permission bits and hardware exception handling

With the permission bits set to R W M (read, write, and modify are not allowed), if the application software attempts to read the Array pointer's address, a memory access violation exception is raised. The OSFA hardware detects a memory access violation and calls microkernel to handle the exception. The system developer could include a microkernel driver to handle the memory exception and even allow reading the address contained in the pointer Array. This would be considered poor coding style, violate the security layer hierarchy and be an open door for hacking.

For example, a high-performance algorithm cannot be proved to be absolutely numerically stable. A second moderate performance algorithm is known to be numerically stable is also running. If the high-performance algorithm raises an exception, the second algorithm is already running and can take over immediately.

5.5 Practical Permission Bit Architecture

Some possible OSFA access levels are shown in Table 4. The access levels are set by cache permission bits. Access levels are specific to the OSFA. OS rings are similar; however, OS rings already have an accepted definition and functions. If we have a trusted application, we can give the application higher-level privileges than an untrusted OS as illustrated in the right-hand column. The secure microkernel in access layers 0 and 1 has exclusive access to the hardware permission bits. No other layers can access the file permission bits.

Table 4 Some possible OS friendly microprocessor architecture access levels

Object	OS friendly microprocessor architecture access levels				
Microkernel (boot)	0	1	n/a	n/a	0
Microkernel	0	0	0	0	0, 1
Microkernel drivers	1	1	1, 2	1, 2	2, 3
Hypervisor	n/a	2	3, 4	3	4, 5
Thick OS	2	3	5, 6, 7	4, 5, 6, 7	7, 8, 9
OS libraries (DLLs)	2	3	7	7, 8	10, 11
OS drivers	2	4	8	8, 9	12, 13
Applications	3	5	9	10, 11	6

Trust Level	Complete	🟩	🟨	🟧	🟫	Untrusted

To illustrate some simple computer security examples, we limit the number of access levels to 4. We can easily increase to 8 or more for a full-featured architecture. The memory cost for the permission bits can be significant. For example, with 8 access levels, 8 permission bits, and 16 bits for the memory type, we would need an additional 96 bits for each cache block. For a 1-kiloword cache bank, this is not significant; however, for each memory cell an additional 96 bits is large. The OSFA Version 2 uses a 1-kiloword permission bit lookup table in a cache bank, which only requires 10 bits. Each memory cell would require 8 bits for layer_0, 8 bits for layer_1, and 10 bits for the cache lookup table (26 bits total), which is much more practical. Layer_0 and layer_1 permission bits are included for each memory cell for high-speed microkernel access. A larger cache bank is possible; however, at some point becomes unpractical like the 96 bits per cell described previously.

The access level for mobile code is set by a trusted certificate authority and verified by the microkernel. Mobile code with an unverified (untrusted) certificate is deleted.

An example, 4-layer permission bit architecture for the OSFA, is illustrated in Fig. 22. To reduce the complexity for describing the architecture's features, we have limited the access layers to 4. As shown in Table 4, more access layers are easy to define. We consider 4 layers the minimum number of access layers for the OSFA. A practical number of access layers is around 8. Section 6 covers some computer security examples for the example architecture described in Fig. 22.

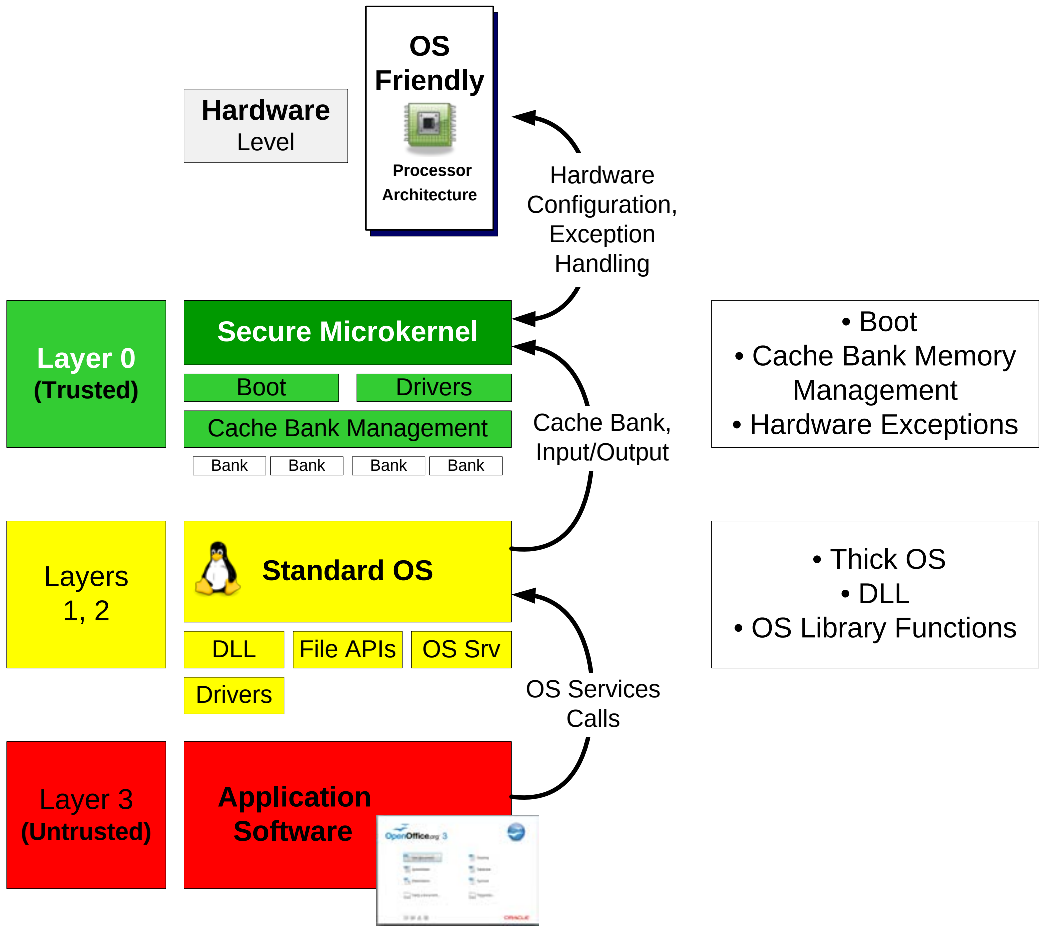


Fig. 22 Example 4-layer architecture

5.6 OS Friendly Microprocessor Architecture Version 2: Practical Cache Bank Architecture

An example practical cache bank and permission bit architecture is described in Fig. 23. A possible 256-bit cache bank header with 5 defined fields is shown. A larger, more complex cache bank header would be easy to define. A memory type field describes the type of memory contained in the cache bank. The PID field describes the "owner" of the cache bank. A microkernel field is defined for additional microkernel control over the cache bank. The undefined field may contain additional microkernel settings, or the OS may call the microkernel to include OS-related cache bank information in the undefined field. Each cache bank contains 256 bits (cache bank header), 64 bits per memory cell, and 32 permission bits per memory cell. For a 1-kiloword (8-KB) memory cache bank, 12 KB of memory are required. For a 4-kiloword (32-KB) cache bank, 48 KB of memory are required. For the example processor, we use 1 kiloword (8 KB) for the cache size. A good compromise for an actual microprocessor would be around 8–64 kilowords.

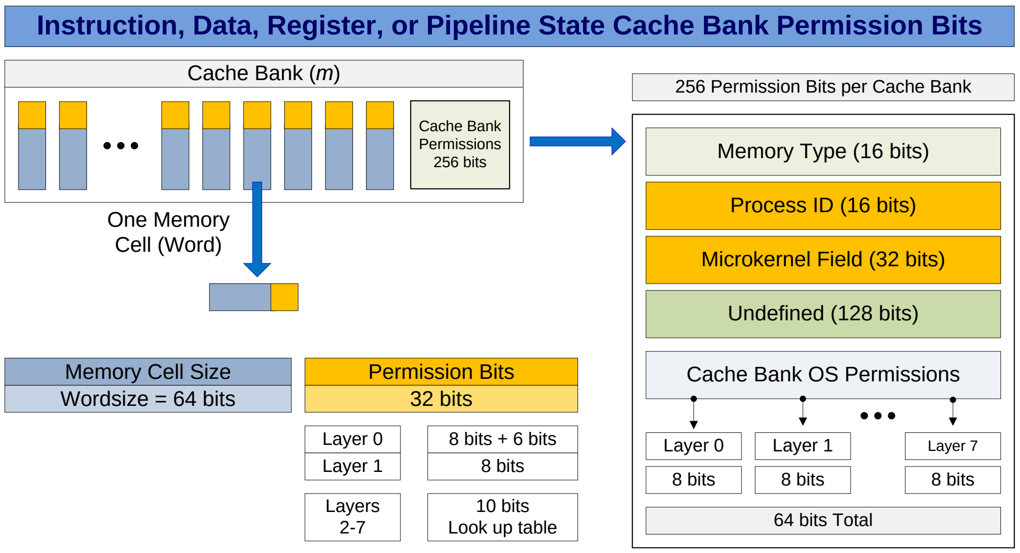


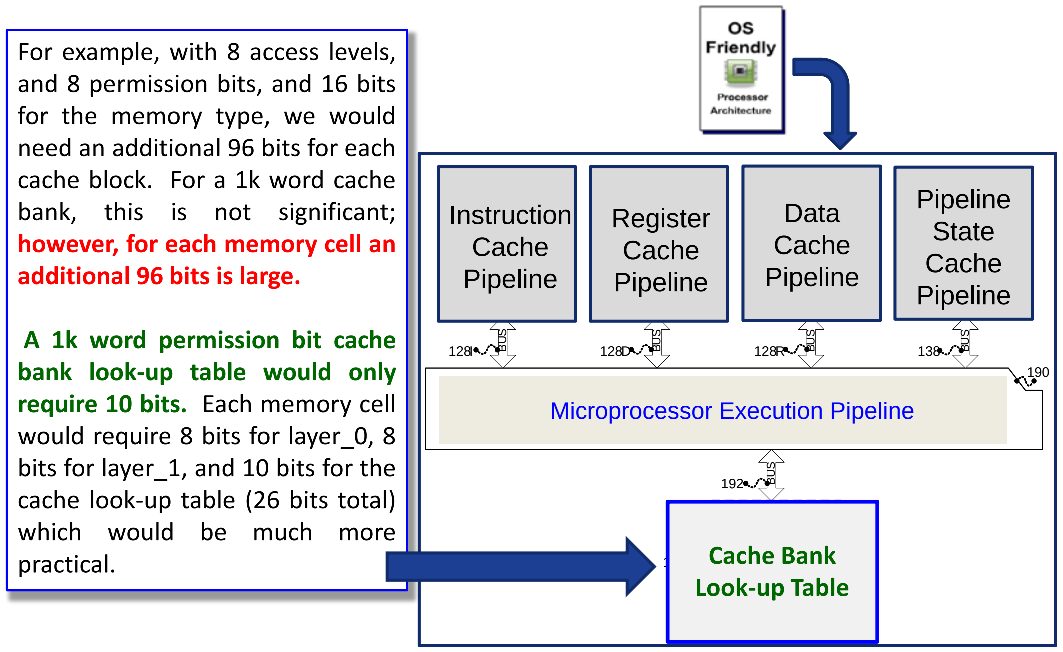
Fig. 23 Practical permission bit and cache bank architecture

5.6.1 OS Friendly Microprocessor Architecture Version 1 Permission Bit Limitations

The memory required for OSFA Version 1 permission bits is large. Most processes (tasks) will only require a few different permission bit settings. For the OSFA Version 2, we implement a cache bank lookup table with 10–16 address bits. This provides a lookup table with 1024–65,536 entries. With a 10-bit lookup table, we can get by with only 32 bits per memory cell to hold the permission bits. Without the lookup table, 64–96 bits per memory cell or more would be required.

5.6.2 OS Friendly Microprocessor Architecture Version 2 Permission Bit Cache Bank Architecture

Figure 24 shows the OSFA Version 2 permission bit cache bank look up table. A 10-bit lookup table provides for 1024 different permission bit settings. Version 1 without the lookup table would require 64–96 bits per memory cell. A 16-bit permission bit cache bank lookup table would provide for 64k of unique permission bit settings. As illustrated in Fig. 23, a 64k cache bank lookup table would only require 38 bits per memory cell for the cache bank permission bits.



For example, with 8 access levels, and 8 permission bits, and 16 bits for the memory type, we would need an additional 96 bits for each cache block. For a 1k word cache bank, this is not significant; **however, for each memory cell an additional 96 bits is large.**

A 1k word permission bit cache bank look-up table would only require 10 bits. Each memory cell would require 8 bits for layer_0, 8 bits for layer_1, and 10 bits for the cache look-up table (26 bits total) which would be much more practical.

Fig. 24 Cache bank permission bit lookup table

5.7 Microkernel, OS, and Application Cache Banks Organization

The OSFA cache bank organization, for the 4-layer model introduced in Figs. 22 and 23, is described in Fig. 25. There are 3 groups of cache banks: microkernel (layer 0), OS (layers 1 and 2), and application software (layer 3). The microkernel controls and configures all cache bank permission bits. The microkernel has full access to the OSFA's permission bits and it is completely trusted. At the hardware level, all cache banks are the same. Only the instructions in the instruction cache banks are executed. The data, register, and pipeline state cache banks are not connected to the instruction decode block in the execution pipeline and cannot be executed.

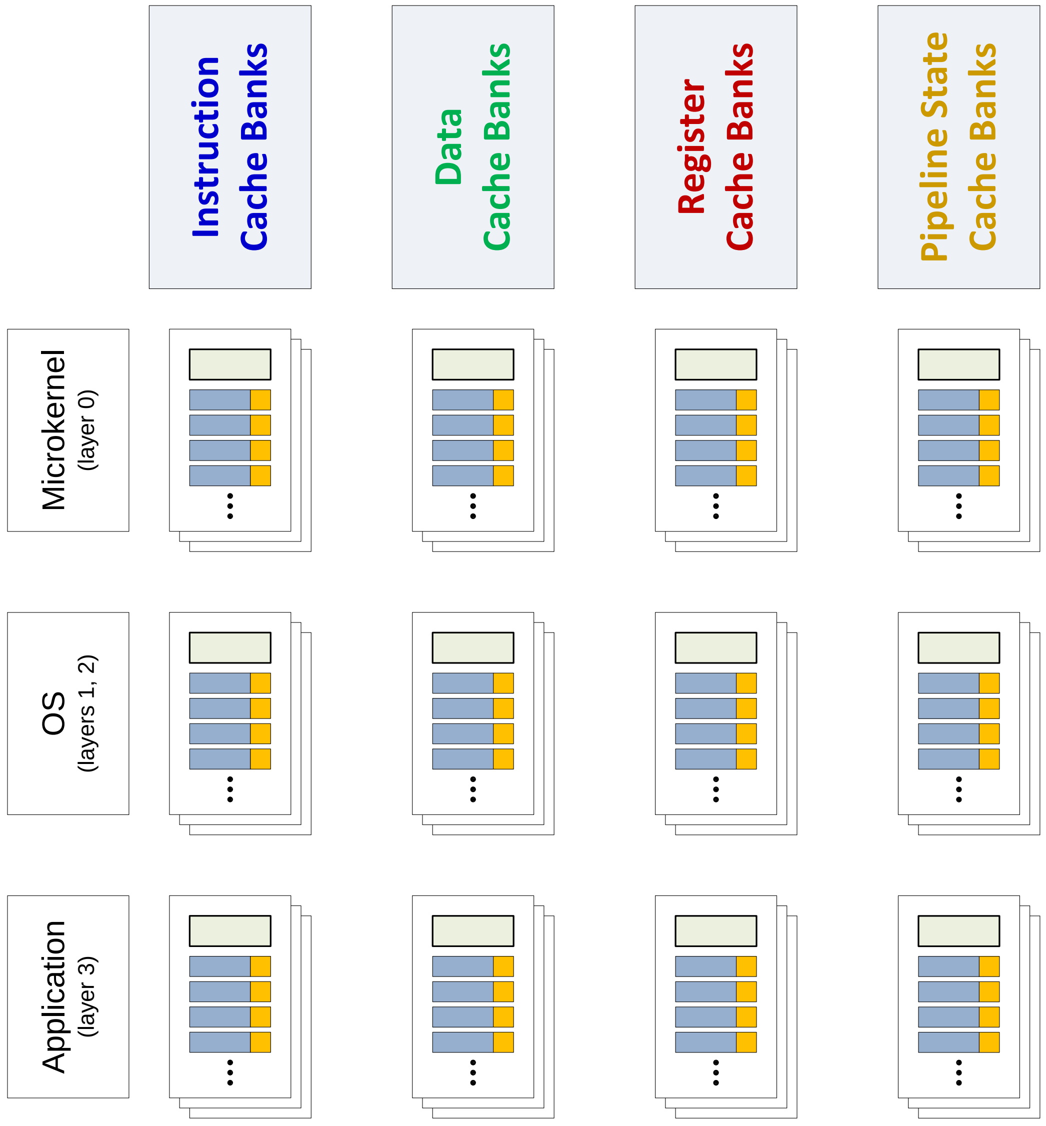


Fig. 25 OS friendly microprocessor architecture cache bank organization

5.8 Process Level Cache Bank Operations

Figure 26 presents a simplified example of microkernel cache banks and cache bank contents. Figure 26 does not present the details for a complete microkernel. The 4 cache bank pipelines, instruction, data, register, and pipeline state, are completely separated. The contents from one cache bank type cannot be copied to another cache bank type. For example, data placed on the data stack cannot be accessed by the instruction cache bank pipeline. Two threads, `thread_0` and `thread_1`, are contained in the instruction cache bank set. Only the instruction cache bank pipeline connects to the OSFA's instruction decode stage in execution pipeline. A third cache bank is used as a return function call stack for the microkernel.

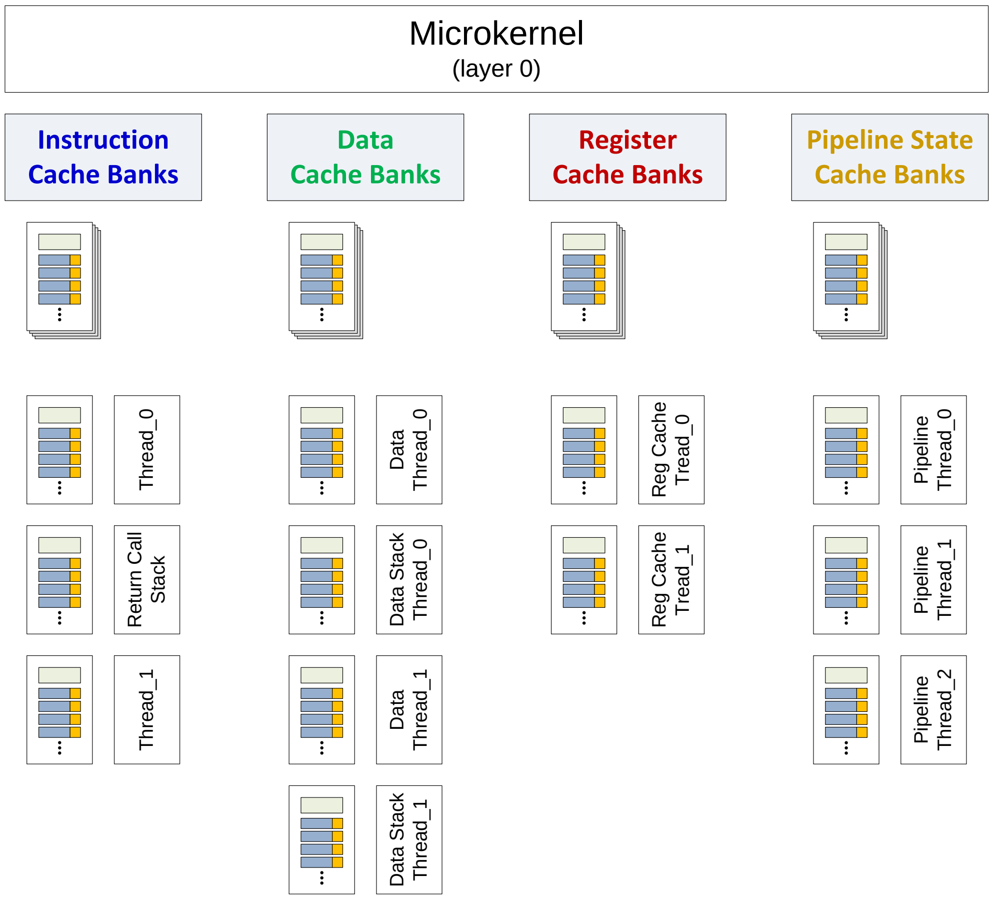


Fig. 26 Microkernel cache bank organization

Data cache banks contain data in use by `thread_0` and `thread_1`. Each thread may be assigned its own stack space in the set of data cache banks. The separation of thread stack areas also enforces isolation between threads. Each thread, `thread_0` and `thread_1`, has its own set of registers in the set of register cache banks. The pipeline state cache banks contain the latch states from the microprocessor's execution pipeline in Fig. 5. The pipeline state cache banks allow for very rapid context switching. Section 4.2 and Fig. 6 describe cache banks and context switches. Context switches for threads can be as fast as 1 CPU cycle using the OSFA.[1]

The instruction, data, register, and pipeline state cache banks in Fig. 27 for the OS and applications software have the same hierarchy as the microkernel. The microkernel configures all cache banks and permission settings. The OS and applications software are "clients" to the microkernel. The microkernel manages all permission bit settings, hardware exceptions, I/O, memory management, and DMA settings for the OSFA. The cache bank architecture provides high-speed context switches, and very efficient I/O.

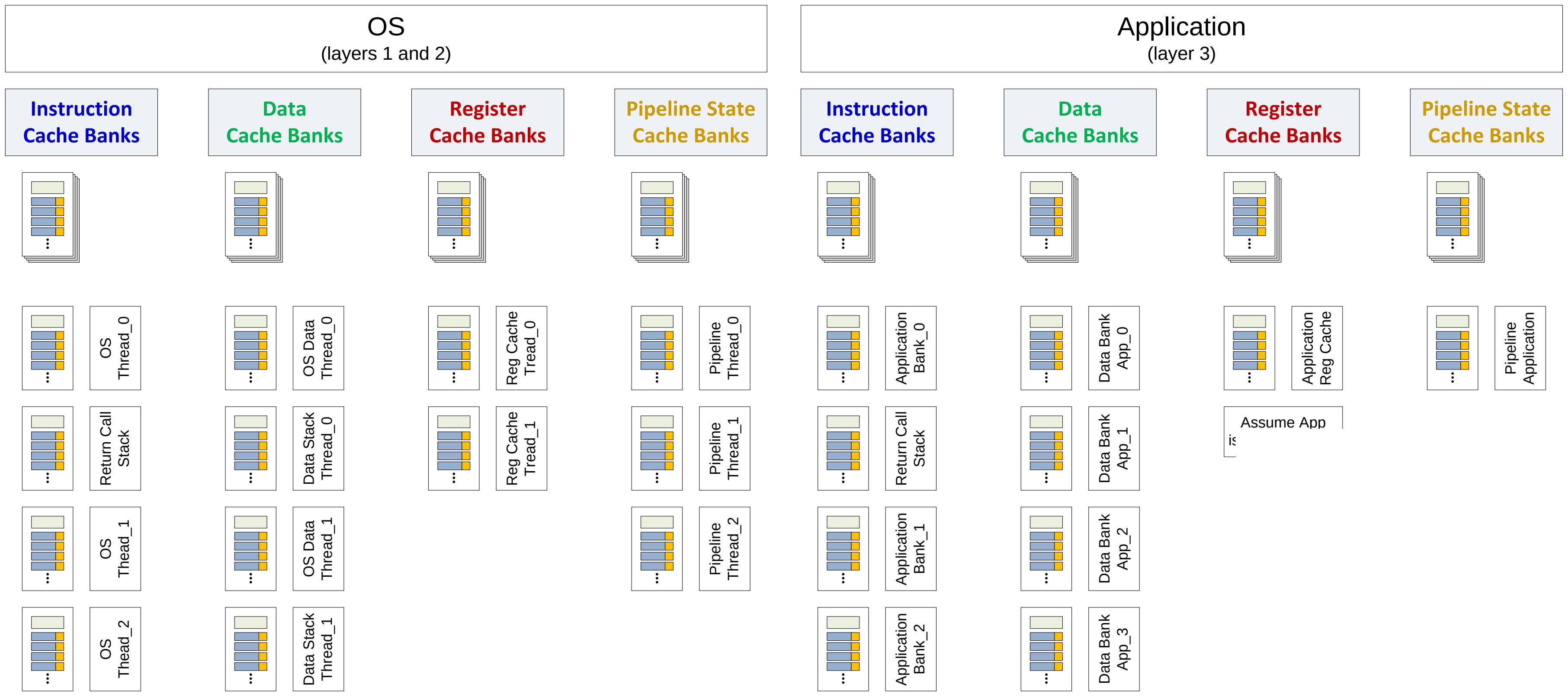


Fig. 27 OS and application cache bank organization

5.9 Cache Bank I/O Example

We present an example in Fig. 28 showing the interaction between microkernel, OS, and applications software for the USB I/O example discussed in Fig. 18. We follow the cache bank organization and discussion found in Sections 5.7 and 5.8. The software application in `Application Bank_0` is executing the machine code equivalent of the high-level instruction `Port = OS.Create_I/O.Port( ● )`. The OS transfers the call to the microkernel by `OSPort = MK.Create_I/O.Port( ● )`. The microkernel creates a pointer to a data cache bank. The application program has permission to write data into the cache bank; however, it cannot access the contents of the pointer (register). The pointer (register) contains the memory address of the cache bank.

Once the data have been placed in the I/O bank for `App_0`, the application calls the OS. The OS calls the microkernel, which simply enables a direct memory access controller starting at the address found in the pointer `Port`.

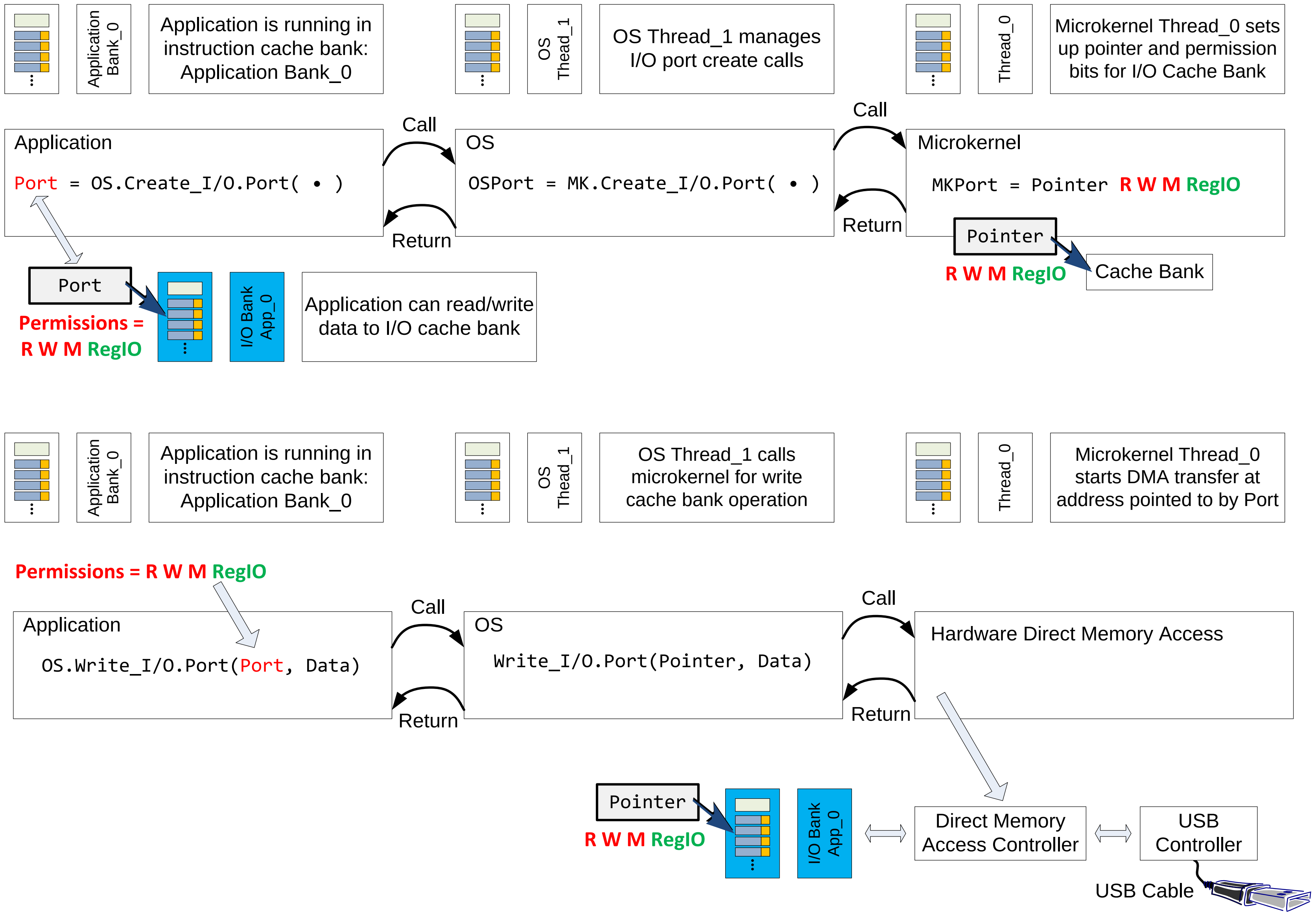

Application Bank_0
Application is running in instruction cache bank: Application Bank_0
OS Thead_1
OS Thread_1 manages I/O port create calls
Thread_0
Microkernel Thread_0 sets up pointer and permission bits for I/O Cache Bank
Application
Port = OS.Create_I/O.Port(•)
Call
Return
OS
OSPort = MK.Create_I/O.Port(•)
Call
Return
Microkernel
MKPort = Pointer R W M RegIO
Pointer
R W M RegIO
Cache Bank
Port
Permissions = R W M RegIO
I/O Bank App_0
Application can read/write data to I/O cache bank
Application Bank_0
Application is running in instruction cache bank: Application Bank_0
OS Thead_1
OS Thread_1 calls microkernel for write cache bank operation
Thread_0
Microkernel Thread_0 starts DMA transfer at address pointed to by Port
Permissions = R W M RegIO
Application
OS.Write_I/O.Port(Port, Data)
Call
Return
OS
Write_I/O.Port(Pointer, Data)
Call
Return
Hardware Direct Memory Access
Pointer
R W M RegIO
I/O Bank App_0
Direct Memory Access Controller
USB Controller
USB Cable

Fig. 28 Application writes a cache bank block of data to USB controller

6. Computer Security Examples

To improve system performance, the OSFA implements context switching, cache memory management, and resource permission bits in hardware. This architecture requires a secure boot to properly set up the permission hierarchy.

The goal is to have each task run in its own hardware "sandbox". All access points to control operations are managed by the hardware permission bits. All tasks are running in an environment without any software level access to a control or management level resources. In the 1970s blue box terms, we are hopefully blocking all paths to an in-band signaling channel.

6.1 Buffer Overflow

Buffer overflow attacks are described in Cowan et al.[14] In telephone in-band signaling,[3–4] a blue box[5–6] was the tool needed to control the telephone network. Back in the 1970s, there was no authentication to prevent a prankster from using a blue box. In a buffer overflow attack, the attacker follows a set of steps avoiding any authentication to reach the goal of administrator privileges. A buffer overflow attack is in the same class as in-band signaling—an open door.

> All of these methods seek to alter the program's control flow so that the program will jump to the attack code. The basic method is to *overflow* a buffer that has weak or non-existent bounds checking on its input with a goal of corrupting the state of an adjacent part of the program's state, e.g. adjacent pointers, etc. By overflowing the buffer, the attacker can overwrite the adjacent program state with a near-arbitrary[15] sequence of bytes, resulting in an arbitrary bypass of C's type system[16] and the victim program's logic.[14]

In Fig. 29, a running process executes the machine code equivalent of `printf("ABCDEFGHIJKLMNO");`. The string is placed on the stack and the OS function `printf( )` is called. The return address is placed on an instruction stack. The executing process can only place data on the "data stack". The process does not have any access to the instruction stack. The return address is not contained on the "data stack". Overwriting the data stack cannot modify the return pointer on the instruction stack. The OS library call will need to check the length of the string. If the stack was overwritten, the `printf( )` call would continue printing characters until a null string character was found or when the `printf( )` library function call tries to access memory out-of-bounds, a memory access violation error would occur. The process does not have access to the return address and cannot maliciously modify the return address.

```
Printf("ABCDEFGHIJKLMNO");
// Place "ABCDEFGHIJKLMNO" On stack and Call OS Function //
```

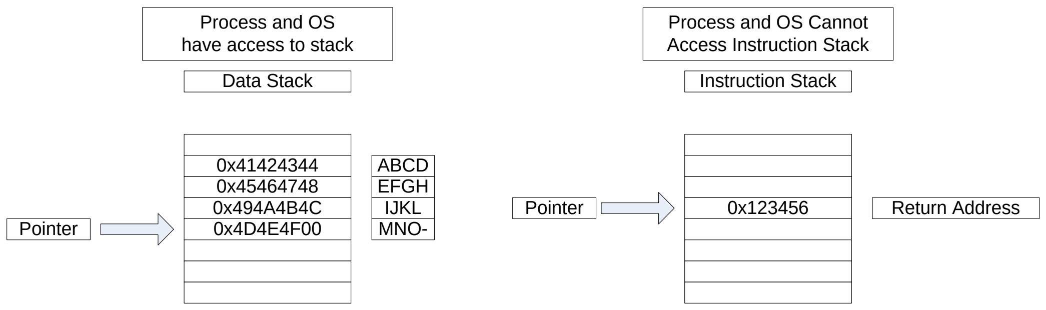


Fig. 29 Process stack example

6.2 Data Execution Exploitation

The OSFA maintains a strong separation between instructions and data. The OSFA (Fig. 5) cache bank memory pipelines (instruction, data, register, and pipeline_state) are not unified. The pipelines are completely separated at the hardware level. The cache bank permission bits (Fig. 12) maintain separation of access layers and permission bits as described in Section 5.2. A practical cache bank permission bit structure is described in Fig. 23. A process level description of permission bits is presented in Section 5.8. Figures 26 and 27 show the strong separation of access layers, cache banks, and processes.

Assume a running malware application has created a data block containing the machine code for a computer virus. To take control of the computer, the malware needs to find an open door to a microkernel level resource and attack the vulnerability. Assume the application, `Application Bank_0`, running in Fig. 27 is malware. The malware does not have any access to its running code in the instruction cache bank pipeline. The malware could create an executable virus in data memory. Assume `Data Bank App_0` contains the machine code instructions for malware. The following paragraphs describe the actions malware would need to take to attack the architecture.

- The malware would need to move, copy, or transfer the data cache bank into an instruction cache bank. Second, the malware would need to access the cache bank permission bits and set the "data" type to executable. The microprocessor architecture shown in Figs. 1 and 5 does not have any logical connection between data and instruction cache bank memory pipelines.

- The malware could place the computer virus in `Application Bank_2` cache bank. The cache bank permission settings set the access level for the malware "application" to level 3 (untrusted). The hardware will not allow an access level 3 application to modify a cache bank, or I/O port. Attempting to write to a privileged resource would instantly raise a hardware memory access violation. As illustrated in Fig. 28, the malware could call the OS and request an I/O port. The port address is contained in a register (for example R10). R10 has access permission bits set to R W M **`IRegIO`**. As illustrated in Fig. 21, any attempt to read the address contained in Register R10 results in a hardware-level exception.

The architecture's permission bits, cache bank memory type, and cache bank permission bits enforce a strong separation between access levels. The conventional von Neumann computer architectures present numerous opportunities for malware to take control of what should be a trusted resource. Avoid allowing the malware any in-band signaling channels to exploit an attack.

6.3 "Low-Level Driver" Protection

We present a low-level driver example for an Ethernet interface. This example follows the USB and open port example found in Fig. 28. A high-level description of an Ethernet frame[17] is found in Fig. 30. The Ethernet frame has a maximum length of about 1518 bytes. One Ethernet frame would fit in an OSFA 512-word (2 KB) cache bank.

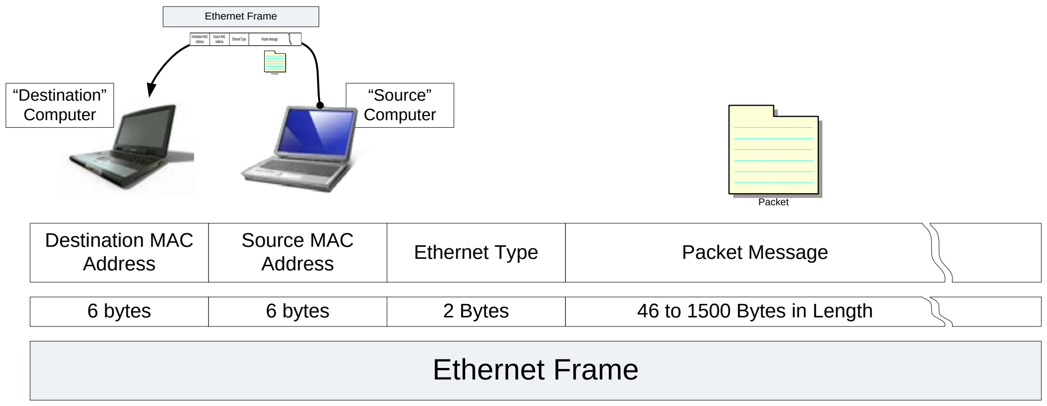


Fig. 30 Ethernet frame

Figure 31 illustrates an OSFA cache bank configured to hold an Ethernet packet. The application software calls the OS, which calls the microkernel to create a pointer to a cache bank. The application then calls OS and microkernel to configure the cache bank memory type as Ethernet. The application calls to set the source and destination MAC addresses; the microkernel then checks to see if the MAC addresses are valid. The source and destination MAC addresses are stored in the cache bank as R W M (not allowed by the application and OS). The source and destination addresses are trusted because the permission bits block access to the application and OS. The microkernel checks the packet length and then configures the DMA controller to transfer the packet to the Ethernet controller. The cache bank architecture and permission bits provide a general interface for all I/O operations for the OSFA.

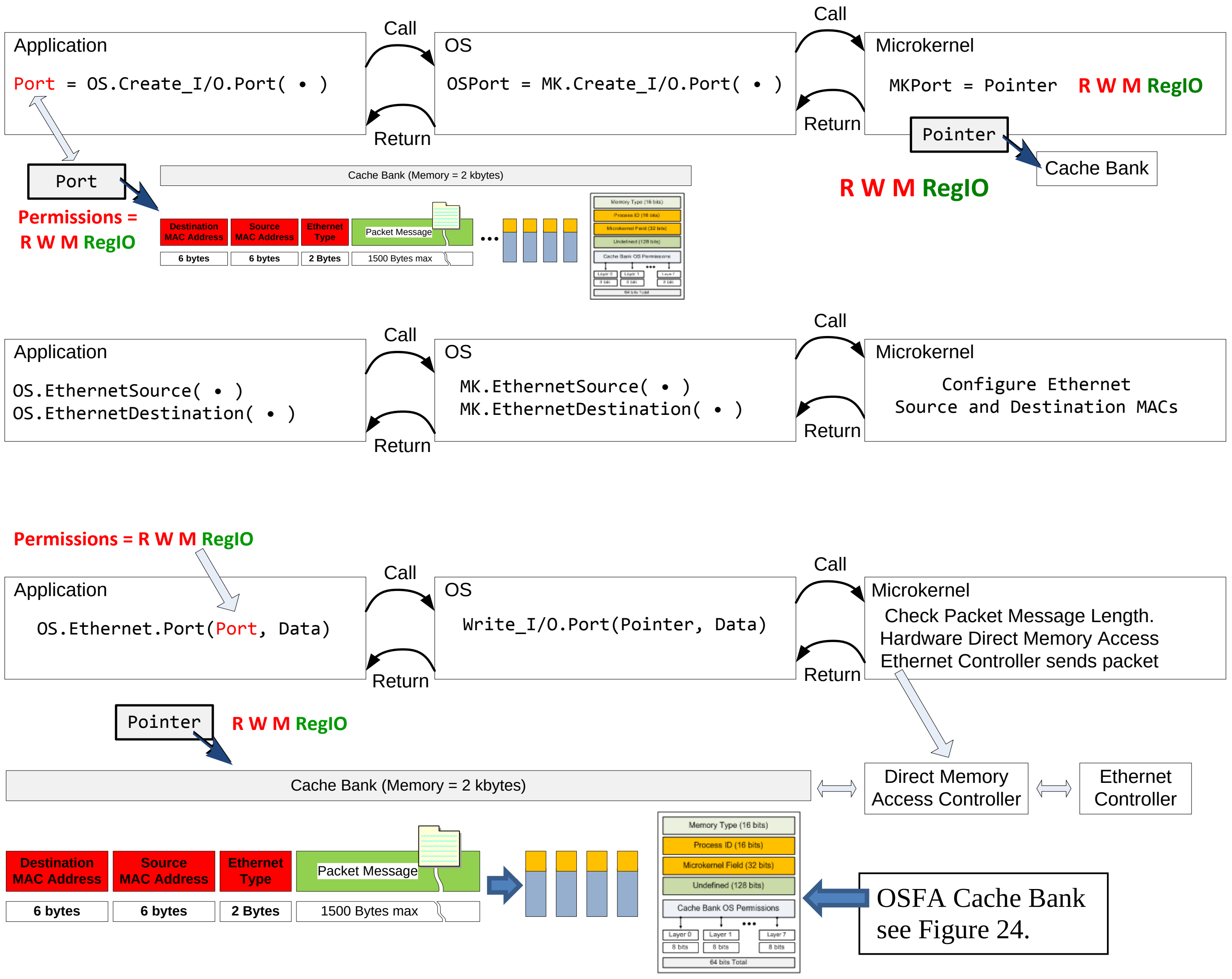


Fig. 31 Cache bank and Ethernet frame example

6.4 Control Information Protection

Computer security examples in Section 6 describe how control information is isolated from OS and application software access levels. All system resources are managed by the microkernel. The microkernel can give the OS access to system resources like memory; however, the OS is limited to only the resources isolated from the microkernel. If an application is trusted more than the OS, the microkernel could be configured to isolated the trusted application from an untrusted OS.

6.5 Debugging Traps

Program debugging at the register level can use the **R** W M permission bits (Read = allowed, Write = not allowed, and Modify = not allowed) to trap all writes made to a memory address or register (Fig. 32). This allows for hardware-level debugging with no performance overhead at the software level until a write occurs.

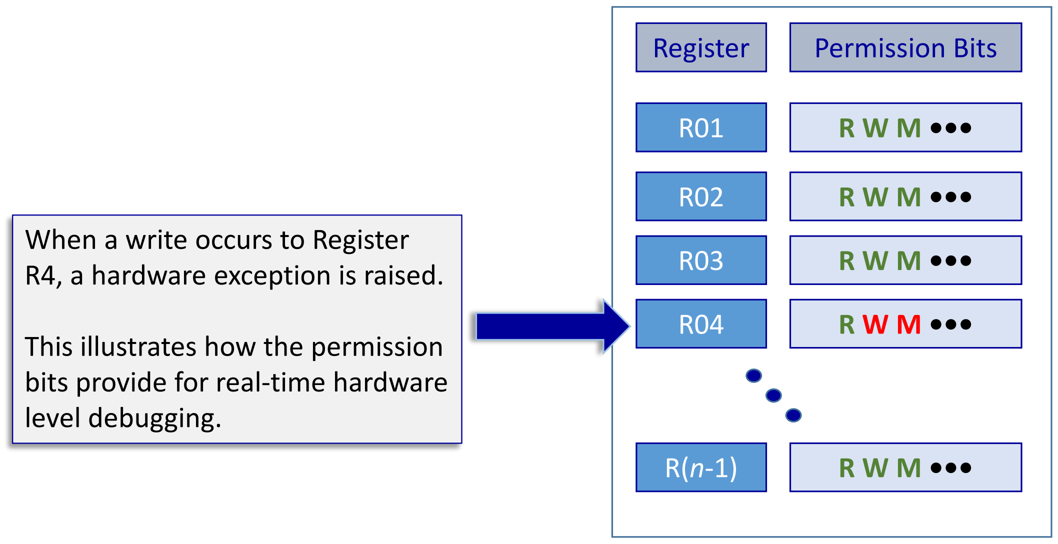

When a write occurs to Register R4, a hardware exception is raised.
This illustrates how the permission bits provide for real-time hardware level debugging.
Register
Permission Bits
R01
R W M •••
R02
R W M •••
R03
R W M •••
R04
R W M •••
R(n-1)
R W M •••

Fig. 32 Real-time debugging trap example

6.6 Hardware Features for Hypervisor

Recent research work has suggested that a fully hardware level (level 0) hypervisor might be possible. A Lynx Technologies white paper [18] states that a level 0, chip level hypervisor would not require OS support. All required OS features are implemented at the hardware level. The goal of creating a type 0 hypervisor may not be possible.[18–20]

The OSFA offers features that would help develop a future level 0 hypervisor. The cache bank and permission bit hierarchy offers hardware-level features needed to create a sublevel 1 hypervisor. For interrupt driven threads, zero overhead context switching is possible with the current OSFA architecture. More hardware features are required to come closer to a level 0 hypervisor.

6.7 Architecture Issues

Proprietary OS uses digitally signed drivers to reduce the chances of malware having easy access to protected OS resources. The OSFA has more complex requirements for digitally signed files. There are computer security trade-offs between a unified external memory (combining instructions, register, data, and pipeline state cache banks) and separate internal cache memories. The same issues are present with unified external mass storage and separate mass storage for the 4 memory pipelines. Figure 33 presents an OSFA system based on Figs. 22–32.

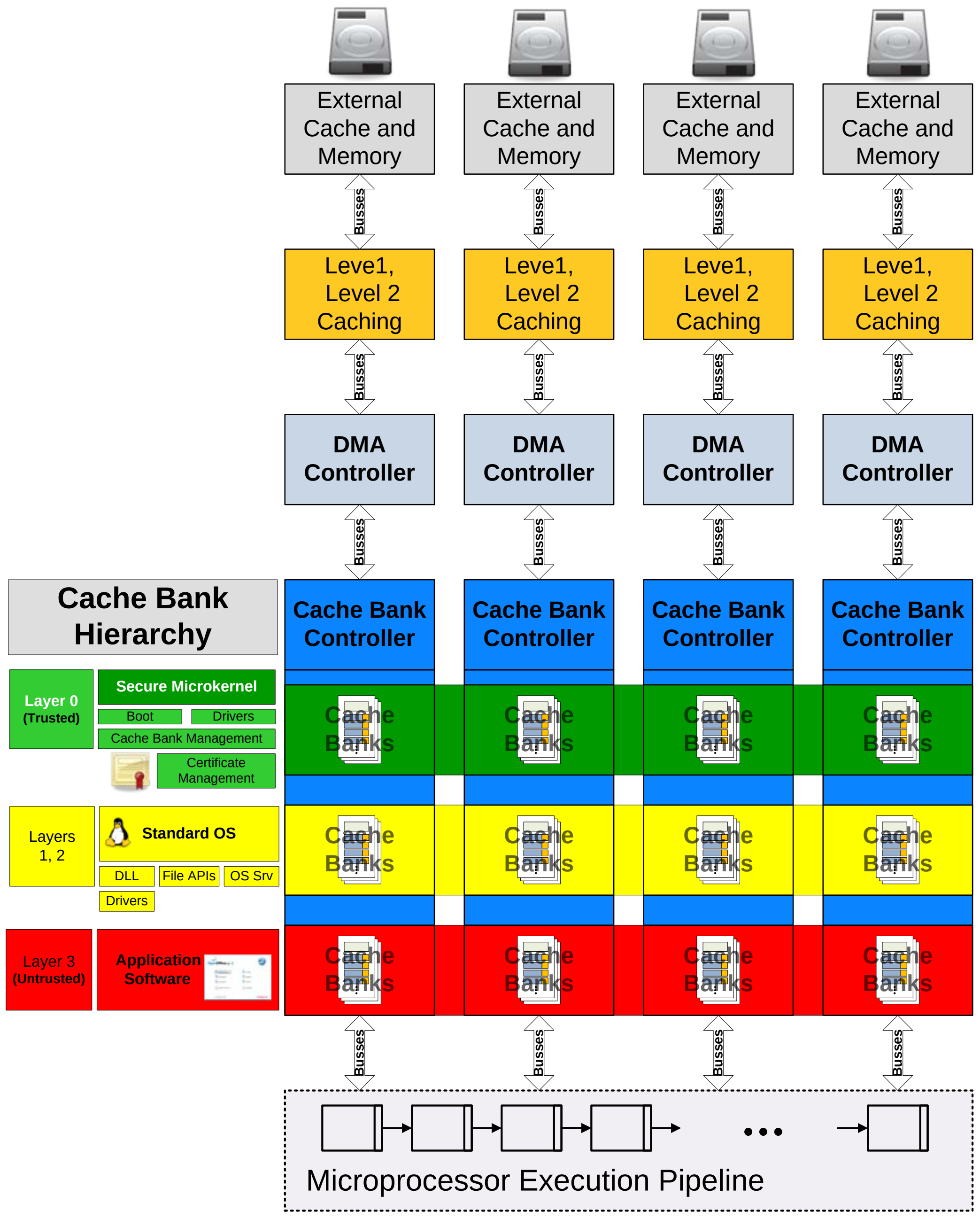


Fig. 33 OS friendly microprocessor architecture: software and hardware hierarchy

7. Conclusion

We have presented an introduction to the hardware and software hierarchy for the OSFA and described hardware-level computer security features. The cache bank memory pipeline architecture and permission bits provide features to balance the complexities of hardware, software, and computer security.

8. References

1. Jungwirth P, La Fratta P, inventors; US Army is assignee. OS Friendly Microprocessor Architecture. United States Patent 9122610. 2015 Sep.

2. Jungwirth P, inventor; US Army is assignee. OS Friendly Microprocessor Architecture: Hardware Level Computer Security. US Provisional Patent Application. 2016 Apr.

3. Weaver A, Newell N. In-band single frequency signaling. Bell System Technical Journal. November 1954;33(6):1309–1330. https://archive.org /details/bstj33-6-1309.

4. Breen C, Dahlbom C. Signaling systems for control of telephone switching. Bell System Technical Journal. November 1960;39(6):1381–1444. https://archive.org/details/bstj39-6-1381.

5. Wikipedia: Blue box. [place unknown]: Wikipedia [updated 2016 Sep 2; accessed 2014 June]. http://en.wikipedia.org/wiki/Blue_box.

6. Computer History Museum. Artifact details: Wozniak's blue box. Mountain View (CA): Computer History Museum [updated 2017; accessed 2014 Dec]. http://www.computerhistory.org/collections/catalog/102713487.

7. Wikipedia: Caller ID spoofing. [place unknown]: Wikipedia [updated 2016 Nov 13; accessed 2014 Dec]. http://en.wikipedia.org/wiki/Caller_ID_spoofing.

8. FCC: Caller ID spoofing. Washington (DC): Federal Communications Commission [accessed 2014 Dec 29]. http://www.fcc.gov/guides/caller-id-and-spoofing.

9. Vangal S, Anders MA, Borkar N, Seligman E, Govindarajulu V, Erraguntla V, Wilson H, Pangal A, Veeramachaneni V, Tschanz JW et al. 5-GHz 32-bit integer execution core in 130-nm dual-V_T CMOS. Solid-State Circuits. Nov. 2002;37(11):1421–1432.

10. Mudge T. Power: a first-class design arch constraint. Computer. Apr. 2001;34(4):52–8.

11. Fillpot M. Understanding Linux file permissions. San Francisco (CA): Linux.com; 2010 May 18 [accessed 2014 Dec 8]. http://www.linux.com/learn /tutorials/309527-understanding-linux-file-permissions.

12. Bacon J. Getting to grips with Linux permissions: do it with permission. Linux Magazine. 2000 Oct; 1:72–74. http://www.linux-magazine.com/Issues/2010 /120/Beginners-File-Control.

13. Wikipedia: File system permissions. [place unknown]: Wikipedia [2017 Jan 31; accessed 2014 Feb] https://en.wikipedia.org/wiki/File_system_permissions.

14. Cowan C, Wagle P, Pu C, Beattie S, Walpole J. Buffer overflows: attacks and defenses for the vulnerability of the decade. SANS 2000 (System Administration, Networking, and Security) Conference; 2000 Mar 21–28; Orlando, FL. p. 1–11. http://www.cs.utexas.edu/~shmat/courses/cs380s_fall09/cowan.pdf.

15. lsap.org. Linux Security Audit Project. c2017 [accessed 2017 Mar 24] http://lsap.org/.

16. Arcangeli A. xterm exploit. Bugtraq mailinglist. 1998 May 8. http://geek-girl.com/bugtraq/, .

17. Morgan D. Ethernet basics. Dallas (TX): Southern Methodist University; 2009 [accessed 2017 Mar 28]. http://homepage.smc.edu/morgan_david/linux/n-protocol-09-ethernet.pdf.

18. Lynx Software Technologies. The rise of the type zero hypervisor. San Jose (CA): Lynx Software Technologies; 2012 July 18 [accessed 2017 Mar 28]. http://www.lynx.com/the-rise-of-the-type-zero-hypervisor.

19. Wikipedia: Hypervisor. [place unknown]: Wikipedia [2017 Feb 6; accessed January 2015]. http://en.wikipedia.org/wiki/Hypervisor.

20. Beaver S, Haletky E. Type 0 hypervisor - fact or fiction. Austin (TX): The Virtualization Practice; [accessed 2012 July 26]. http://www.virtualization practice.com/type-0-hypervisor-fact-or-fiction-17159/.

List of Symbols, Abbreviations, and Acronyms

ARL	US Army Research Laboratory
CPU	central processing unit
I/O	input/output
ID	identification
IRegIO	Index_Register_I/O
OS	operating system
OSFA	Operating System Friendly Microprocessor Architecture
PID	process identification
RFS	Register File Set
USB	Universal Serial Bus

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
2 (PDF)	DIRECTOR US ARMY RESEARCH LAB RDRL CIO L IMAL HRA MAIL & RECORDS MGMT
1 (PDF)	GOVT PRINTG OFC A MALHOTRA
1 (PDF)	DIR USARL RDRL CIH S P JUNGWIRTH